TAMPEREEN TEKNILLINEN YLIOPISTO

VILLE AHO

PITKÄN ELINKAAREN OHJELMISTON KEHITTÄMINEN

Diplomityö

Tarkastaja: professori Hannu Jaakkola

Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
7. marraskuuta 2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
AHO, VILLE: Pitkän elinkaaren ohjelmiston kehittäminen
Diplomityö, 48 sivua
Toukokuu 2013
Pääaine: Ohjelmistotekniikka
Tarkastaja: professori Hannu Jaakkola
Avainsanat: Ohjelman evoluutio, ohjelmiston elinkaari, ylläpidettävyys, jatkokehittäminen, uudistaminen, konfiguraation hallinta, versiointi, versiohallinta, arkkitehtuurityyli

Ohjelmiston elinkaareen kuuluu monia erilaisia vaiheita ja muutoksia. Pitkä elinkaari asettaa haasteita niin määrittelyllisesti kuin teknisestikin. Tässä työssä pyritään selvittämään, miten pitkään olemassa ollutta ohjelmistoa voidaan kehittää mahdollisimman tehokkaalla tavalla. Tavoitteena on löytää tekniikoita ja työvälineitä, joilla kehitystyötä voidaan viedä tehokkaasti eteenpäin.

Työ jakautuu neljään osa-alueeseen. Ohjelman nykyisen rakenteen tarkastelussa luodaan käsitys ohjelmassa käytetyistä arkkitehtuurityyleistä. Evoluution hallinnassa tutkiskellaan sitä, miten ohjelmaa muutettaessa pystytään hallitsemaan muutostilanteet, varmistumaan lopputuloksesta ja millaisia riskejä muutokset sisältävät. Työkalujen tarkastelussa pyritään löytämään nykyisistä työkaluista hyödyntämismahdollisuudet. Lisäksi kartoitetaan mitä vielä tuntemattomia apuvälineitä on mahdollista hyödyntää. Tuotteen koostamisessa perehdytään tuotteen rakenneosien hallinnointiin eri ohjelmaversioiden välillä. Samalla tarkastellaan millä käytännön toimenpiteillä ja tekniikoilla koostamisprosessia hallinnoidaan.

Tutkielman tuloksena havaitaan, että arkkitehtuurityylien ymmärtäminen ja noudattaminen selkeyttävät ohjelman luettavuutta sekä pienentävät virheiden todennäköisyyttä ohjelmaa muutettaessa. Toiminnan prosessimalli on tuotava entistä selkeämmin esiin jokapäiväisessä työssä, jotta sitä voidaan kehittää paremmin työyhteisöön sopivaksi. Ylläpito ei ole vain virheiden korjaamista. Siihen kuuluu myös ohjelmaan liittyvän dokumentaation parantaminen. Ominaisuuksia lisättäessä tai muutettaessa on ymmärrettävä selvästi niiden vaikutusalue. Oleellista on myös selvittää, millaista hyötyä ominaisuuksilla saadaan, mitä riskejä niihin liittyy ja mihin ohjelmaversioon muutos on tarkoitus toteuttaa. Henkilöresurssien käyttö on tehokasta silloin, kun se on suunniteltu joustavaksi. Työkaluja on pystyttävä hyödyntämään entistä tehokkaammin. Siinä voidaan käyttää apuna tiimin sisäistä dokumentaatiota ja muuta tiedonvaihtoa. Työkaluja voidaan hyödyntää nykyistä enemmän erityisesti ohjelman suorituskyvyn analysointiin. Hajautetun versiohallinnan hyödyntäminen on vasta alussa. Siihen liittyvät toimintatavat edellyttävät vielä kehittämistä. Kirjattujen tehtävien ja niihin liittyvien muutosten linkittäminen tehostavat integraatiotestausta.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
AHO, VILLE: Long life cycle software development
Master of Science Thesis, 48 pages
May 2013
Major: Software Engineering
Examiner: Professor Hannu Jaakkola
Keywords: Software evolution, software lifecycle, maintainability, software development, re-engineering, configuration management, versioning, version control, architectural style

Many different types of phases and changes are involved with the lifecycle of software. Long lifecycle puts challenges for both requirements and technologies. In this thesis goal is to sort out most efficient ways to develop long existed software. Object is to discover techniques and tools by which it is possible to carry out development work efficiently as possible.

The thesis is divided into four sections. In the analysis of software's current structure, goal is to create understanding architecture styles used throughout in software. In evolution control studying aim is in the event of change and managing those events, verifying outcome and evaluating the risks of changes to be made. Studying tools is about exposing utilisation possibilities out of current tools. It is also to survey what other still unknown tools are there to be utilized. Product composition is about managing components between different versions of software. Simultaneously used procedures and techniques of managing composition process are being examined.

As a result of this study, it is being noted that understanding and following of architectural styles clarify software readability and also reduce the risk of errors while making changes to software. The work process model has to be brought out more distinctly in every day work for it to be improved better matching to work community. Maintenance isn't just bug fixing. It also includes improving software related documentation. While adding or changing features, one has to understand target area of those. It's also essential to study what kind of benefit it'll bring, what risks are involved with those and to which version it's meant to be implemented. Use of personnel resources is efficient when it's designed to be flexible. Tools are to be used more efficiently. As a help there can be used teams internal documentation and other data exchange. Tools can be utilised more than now to especially for software's performance analysis. Invocation of distributed version control is just beginning. Procedures relating to it still require development. Registered tasks and linking changes relating to those streamline integration testing.

ALKUSANAT

Aloittaessani tämän diplomityön tekemisen minulla oli selvä käsitys siitä, mitä halusin sillä saavuttaa. Olin silloin ollut työssäni vasta hyvin vähän aikaa ja olin myös melko kokematon alalla. Työn edetessä kokemusta karttui niin päivätyön kuin diplomityönkin avulla. Matkan varrella karttunut tietämys sai minut näkemään tiettyjä asioita toisella tavalla kuin olin alussa ajatellut. Toivon ja oletan, että jatkossakin vielä tulee tilanteita, jotka opettavat yhtä paljon kuin tämän prosessin läpikäyminen.

Tämä työ on ollut alusta loppuun asti hyvin itsenäistä. Aihealueiden rajauksia tehtiin professori Jaakkolan kanssa, joka on myös tämän työn ohjaaja. Projektista karsiutui pois jonkin verran asioita, koska niiden järkevä käsittely olisi vaatinut vielä huomattavan määrän työtä ja samalla muuttanut painotusta. Oma toiveeni on, että tämän työn tuloksien perusteella työyhteisössä käydään laajempaa keskustelua aiheesta sekä tarkastellaan toimintatapoja.

Haluan kiittää diplomityöni ohjaajaa professori Hannu Jaakkolaa asiantuntevasta avusta, jolla työ pysyi uomissaan eikä lähtenyt sivuraiteille. Ilman sitä ohjausta tämä työ olisi tuskin vielä valmis.

Haluan myös kiittää esimiestäni Jari Fastia hänen ajastaan ja mielenkiinnostaan työtäni kohtaan. Toivon, että hänen tähän käyttämänsä aika on ollut sen arvoista. Työkollegoiden tuki on ollut tärkeää niin tämän työn sisällön kannalta kuin senkin, että he ovat innostaneet sen toteuttamisessa.

Ison kiitoksen ansaitsevat myös perheenjäsenet, jotka ovat auttaneet minua useilla eri tavoilla tämän työn valmistumisen eteen. Olen varma, että autoitte vaivojanne säästelemättä.

Porissa 5. toukokuuta 2013

_____

Ville Aho

Sisällys

1 Johdanto 1
2 Tuotteen rakenne 3
2.1 Arkkitehtuuri ohjelmistotuotteessa 3
2.2 Arkkitehtuurityylit 4
2.2.1. Kerrosarkkitehtuuri 4
2.2.2. Tietovarastot 6
2.2.3. Viestinvälitysarkkitehtuuri 8
3 Evoluution hallinta 11
3.1 Prosessimallit 11
3.2 Ylläpidettävyys 12
3.3 Jatkokehittäminen 15
3.4 Uudistaminen 18
4 Työkalut kehitysympäristössä 22
4.1 Lähdekoodin analysointi ja hallinta 23
4.2 Tietokantojen kehitys 27
4.3 Raportoinnin työkalut 30
5 Tuotteen koostaminen 34
5.1 Konfiguraation hallinnan periaatteet 35
5.2 Versiohallintajärjestelmät 36
5.3 Team Coherence - ja Kiln - versiohallintajärjestelmät 39
5.4 Hajautetun versiohallinnan työmallit 40
5.5 Muutosten seuranta 42
6 Päätelmät 44
Lähteet 47

1 JOHDANTO

Jokaisella ohjelmistotuotteella on omanlaisensa elinkaari. Sitä voidaan pitää myös ohjelmiston evoluutiona, joka koostuu joukosta muutoksia [1, p. 20]. Määrittelyvaiheessa ohjelmistolle laaditut vaatimukset saattavat tarvita päivittämistä luonnollisista syistä, kuten lakien ja säädösten muuttuessa. Tarve niiden päivittämiselle voi tulla myös sidosryhmiltä. Nämä ovat syitä ohjelmaan tehtäville muutoksille ja korjauksille, joita ei vielä välttämättä määrittely- tai suunnitteluvaiheessa pystytä arvioimaan. Muutoksilla ohjelmaan tulee uusia rakenteita vanhojen rakenteiden päälle. Ohjelmakoodiin saatetaan tehdä oikopolkuja, jotka mahdollistavat tietyn toiminnan helpommin tai tehokkaammin, mutta jotka kuitenkin voivat rikkoa alkuperäisen arkkitehtuurin periaatteita. Ilman korjaavia toimenpiteitä on todennäköistä, että jossakin vaiheessa kehittäjien aikaa kuluu enemmän ylläpidollisiin tehtäviin kuin uusien ominaisuuksien luomiseen. [1, p. 18]

Tämän työn lähtökohtana on asemansa yritysten taloushallintajärjestelmien markkinoilla vakiinnuttanut tuoteperhe, joka sisältää neljä eri ohjelmatasoa. Tuoteperheessä on useita erilaisia osioita laskutuksesta toiminnanohjaukseen. Kaikille ohjelmatasoille ei ole kaikkia osioita saatavilla. Pienimmässä tuotteessa osioita on vain yksi, suurimmassa niitä on 26. Joidenkin ohjelmatasojen välillä osioiden toiminnallisuudessa on myös eroavaisuuksia. Käyttäjät voivat hankinnan jälkeen lisätä ohjelmaan erilaisia osia ohjelmatason antaman mahdollisuuksien puitteissa tai päivittää sen isommalle tasolle. Pääasiallinen käyttäjäkunta koostuu pienistä ja keskisuurista yrityksistä, joissa ei ole omaa osastoa tai henkilöstöä hoitamaan tietoteknisiä asioita.

Tässä työssä keskitytään tarkastelemaan menetelmiä, joilla nykyiseen kehitystyöhön saadaan uusia näkökulmia. Työn avulla valmistaudutaan tulevaisuuden haasteisiin ja vaatimuksiin sekä siihen, miten suuressa muutoksessa säilytetään ohjelmakoodiin kertynyt tietämys sovellusalueesta. Lisäksi etsitään mahdollisuuksia sen suhteen, miten kehityksen resursseja siirretään vanhojen ominaisuuksien korjaamisesta uusien kehittämiseen. Kehitystoiminnassa huomioidaan taustalla vaikuttavat pienen tiimin rajoitteet ja hyvin rajalliset resurssit.

Ohjelmiston rakenne ja sen tunteminen on keskeinen osa kehitystyötä. Rakenteen kuvauksilla luodaan käsitys nykytilanteesta ja käytetyistä malleista. Rakennetta tarkastellaan eri arkkitehtuurinäkökulmista laajan käsityksen muodostamiseksi. Sisäisen rakenteen lisäksi tutkitaan, millaisia riippuvuussuhteita ohjelmassa on ulkoisiin komponentteihin. Varsinaista kehitystoimintaa käsitellään teknisenä toteutuksena. Toiminnan ryhmäluonnetta tarkastellaan siltä osin kuin se oleellisesti vaikuttaa tekniseen puoleen, toteutusratkaisuihin ja aikataulutukseen.

Evoluution hallinnan kautta tarkastellaan tuotteen kehityksen ja ylläpidon yleisimpiä toimenpiteitä sekä sitä, miten virheitä tutkitaan ja korjataan. Uusien ominaisuuksien lisäämistä lähestytään valintaprosessin, ongelmanratkaisun ja rakenteen eheyden säilyttämisen kannalta. Ohjelmiston uudistaminen suunnataan tulevien tilanteiden ja suurten muutosten toteuttamiseen ohjaavana toimintana. Siinä on tarkoituksena arvioida vaadittavia toimenpiteitä silloin, kun ohjelman uudistaminen tapahtuu suuressa mittakaavassa.

Tietotaidon lisäksi työkalut ovat suuressa roolissa tuotteen muokkaamisessa. Niiden perusteellinen tunteminen ja oikealla tavalla hyödyntäminen auttavat suoriutumaan työstä paremmin. Kehitysympäristön nykyisten työkalujen lisäksi tarkastellaan muita tarjolla olevia työvälineitä sekä arvioidaan niistä mahdollisesti saatavaa hyötyä.

Versioiden hallinta ja koostaminen muodostavat oman kokonaisuutensa ja ne ovat siten merkittävä osa evoluution hallittavuutta, jotta ohjelmistoa pystytään kehittämään pitkällä aikavälillä. Tuotteen koostamisen yhtenä osana ovat versiohallintajärjestelmät. Siksi niiden yleisten toimintaperiaatteiden tunteminen on oleellista. Sen lisäksi vertaillaan käytössä olevaa keskitettyä versiohallintaa hajautetun mallin edustajaan. Lisäksi työssä perehdytään hajautetun versiohallinnan mahdollistamiin työskentelymalleihin.

2 TUOTTEEN RAKENNE

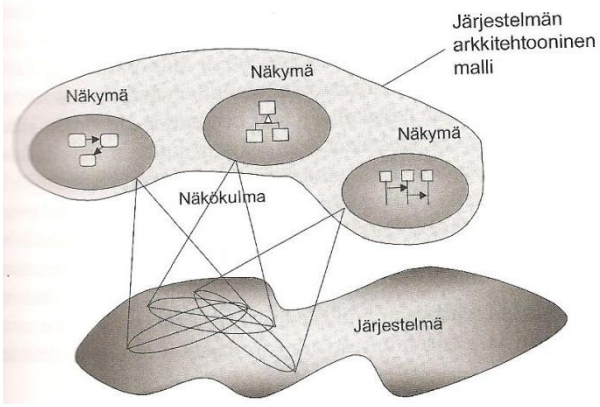
Tässä luvussa tutustutaan ohjelmistojen rakenteeseen käsitteenä. Yleisen katsauksen jälkeen tarkastellaan kohdeohjelman rakennetta. Aivan pienimpiin yksityiskohtiin ei paneuduta vaan keskitytään yleisiin ratkaisuihin ja toteutusmenetelmiin, joita käytetään yleisesti. Kokonaisuuden kannalta tämä luku on pohjustava osio, joka luo käsitystä ohjelmiston nykyisestä rakenteesta.

2.1 Arkkitehtuuri ohjelmistotuotteessa

Jokaisessa hyvin tehdyssä ohjelmassa on rakenteita, joita se noudattelee. Ohjelmista löytyy erityyppisiä rakenteita. Toiset niistä ovat hyvin pintapuolisia ja niiden muuttamisella on merkitystä usein vain tietyssä toiminnallisuudessa. Toisten muuttamisella voi olla ohjelmaan vakauteen tai suorituskykyyn nähden merkittäviä vaikutuksia. [2, pp. 242-243] Ohjelmassa samaa toteutustapaa noudattavia rakenteita voidaan suppeasti käsitellen kutsua sen arkkitehtuuriksi. [3, p. 18]

Ohjelmiston arkkitehtuurin valinnalla on kauaskantoisia vaikutuksia, koska arkkitehtuurin muuttaminen käytössä olevaan ohjelmaan on työmäärän lisäksi erittäin riskialtista. Siksi ohjelmiston määrittelyvaiheessa täytyy olla käsitys siitä, mitä ohjelmalla halutaan ja mitä ei haluta tehdä. Valinta ei välttämättä estä rajauksen ulkopuolelle jääviä asioita, mutta suuntaa kehityksen siten, että niille ei anneta painoarvoa valintoja tehtäessä.

Usein ohjelmisto ei koostu pelkästään yhden arkkitehtuurityylin päälle, vaan se sisältää osia useista arkkitehtuurityyleistä sovellusalueen vaatimusten mukaisesti. Ohjelmassa nähtävien osien voidaan ajatella toimivan tietyn arkkitehtuurityylin mukaisesti tarkasteltaessa ohjelmaa jonkun näkökulman (*viewpoint*) kautta (kuva 2.1.) [3, p. 24].



Kuva 2.1. *Arkkitehtuuri näkymien kautta [3]*.

Järjestelmän arkkitehtuurinen malli syntyy useiden näkymien kautta. Tilannetta havainnollistetaan kuvassa 2.1.

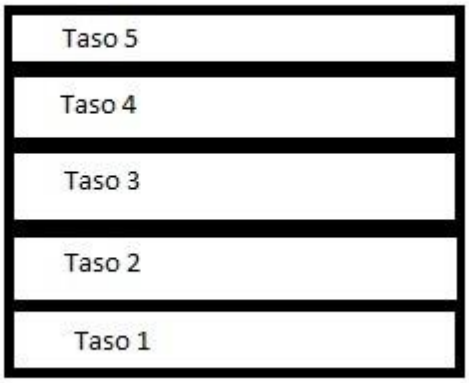
2.2 Arkkitehtuurityylit

Arkkitehtuurinäkymien (kohta 2.1) avulla ohjelmasta voidaan tarkastella pienempää kokonaisuutta. Haluttaessa tarkastella ohjelmaa laajemmalti voidaan hyödyntää arkkitehtuurityylejä. Niihin voidaan liittää tyylistä riippuen erilaisia suunnittelumalleja (*design pattern*) sekä niissä huonosti toimivia malleja (*anti-pattern*). Tämän työn kannalta on perusteltua tarkastella ohjelmia kerros-, tietovarasto- ja viestinvälitysarkkitehtuurien avulla. Kerrosarkkitehtuuri kuvaa enemmän osien järjestystä suhteessa toisiinsa ja tietovarastoarkkitehtuuri rinnakkaisten osien tiedonvaihdon periaatteita. Viestinvälitysarkkitehtuuri puolestaan kuvaa taustajärjestelmän toimintaa.

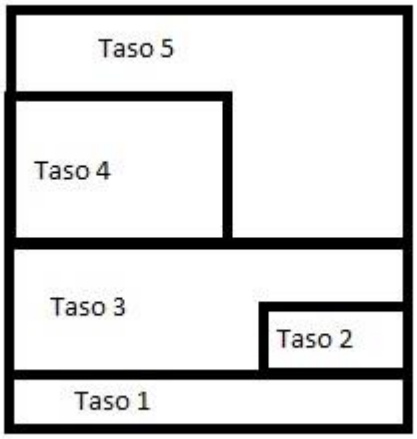
2.2.1. Kerrosarkkitehtuuri

Kerrosarkkitehtuurin periaatteiden mukaisesti ohjelmasta täytyy pyrkiä löytämään komponenttien looginen hierarkia. Paras tapa sen muodostamiseen on lähteä alhaalta ylöspäin. Samalla luodusta kuvauksesta pystytään arvioimaan kerrosten rinnakkaisuutta.

Kerrosarkkitehtuuri kuvaa sellaista rakennehierarkiaa, jossa järjestelmän osat muodostavat loogisesti kerrosmaisen rakenteen. Jokaisen ylemmän tason on tarkoitus yksinkertaistaa sen alapuolella olevan kerroksen toimintaa. Jokainen kerros siis tarjoaa ylemmälle kerrokselle yksinkertaistetun rajapinnan omista ja alapuolellaan toimivista komponenteista. Ideaalisessa versiossa jokainen kerros peittää allaan olevan kerroksen kokonaan (kuva 2.2.). Todellisuudessa täydellinen peittäminen on hyvin vaikeaa, vaatii tekijöiltä enemmän ja tekee järjestelmästä hitaamman tietyissä tilanteissa. Siksi usein kerrosarkkitehtuuria esiintyykin hieman vapaammassa muodossa, jossa alemman tason komponentti tarjoaa rajapintaa useammalle komponentille (kuva 2.3.).



Kuva 2.2. *Ideaalinen kerrosarkkitehtuuri.*



Kuva 2.3. *Yleinen toteutus kerrosarkkitehtuurista.*

Komponenttien järjestys hierarkiassa määräytyy usein oliopohjaisissa kielissä periytyvyyden kautta. Joidenkin komponenttien voidaan ajatella olevan alemmalla tasolla myös kutsusuhteiden kautta. Perusperiaatteena ylemmältä tasolta kutsutaan alempaa tasoa. Alemmalta tasoltakin voidaan kutsua ylempää tasoa, mutta se on hyvä toteuttaa takaisinkutsujen avulla, jotta alempana oleva kerros ei tule riippuvaiseksi ylemmästä.

Käyttöjärjestelmäkerros muodostaa pohjan toiminnalle. Se sisältää yleisen API:n mukaisia toimintoja, jotka ohjaavat kaikista primitiivisimpiä toimintoja kuten piirtämistä ja muistin varaamista keskusmuistista.

Toiseksi alimpana tasona toimii tietokanta. Se ei kutsu ohjelmasta mitään osia. Ainoa toiminta tulee sen aiheuttamien poikkeuksien kohdalla, jolloin ylempi kerros reagoi vikatilanteeseen ja ilmoittaa siitä käyttäjälle. Tietokantakerros on pyritty pitämään hyvin paljon erillään muusta ohjelmasta sen takia, että ei syntyisi vahvaa sidonnaisuutta kyselyiden kieleen.

Tietokantakerroksen yläpuolella on ADO (*Active Data Objects*) kerros. Se on joukko COM (*Component Object Model*) olioita. Kerroksen tehtävänä on huolehtia tietokantayhteydestä ja tiedon noutamisesta. Kerros ei ole riippuvainen sen alapuolella toimivasta tietokannasta, eikä se myöskään ota kantaa SQL-lauseisiin joilla tietoa haetaan. Virhetilanteista ilmoitetaan funktiokutsujen paluuarvoissa.

ADO-kerrosta käyttää pääasiallisesti liiketoiminta-kerros (*Business layer*). Kerros koostuu liiketoiminta-olioista (*Business objects, BO*) [2, p. 277]. Ne ovat sellaisia luokkia, joiden on tarkoitus esittää jonkin todellisen maailman asian ominaisuuksien joukkoa. Liiketoimintaluokan on tarkoituksenmukaista sisältää ominaisuuksien lisäksi toimintaa. Sellaista toteutusta, jossa liiketoimintaluokat eivät sisällä toiminnallisuutta, pidetään aneemisena (*Anemic domain model*) [4]. Kyseisen kaltaista suunnittelumallia voidaan pitää myös antimallina, koska se saattaa johtaa siihen, että samoista asioista joudutaan tekemään monia toteutuksia ja menetetään selkeyden lisäksi ylläpidettävyyttä [4].

Liiketoimintakerroksen yläpuolella toimii palvelukerros (*Middleware*-kerros). Tämän kerroksen päällimmäisenä tarkoituksena on toteuttaa käyttäjälle näkyvät toiminnot. Lisäksi kerros huolehtii tietokannan transaktioista, käyttöoikeuksien tarkistuksesta sekä joissain tapauksissa virheiden ilmoituksista. Osa yksinkertaisimmista ja vähemmän virheherkistä tietokantaoperaatioista voidaan suorittaa myös liiketoimintakerroksella.

Ylimpänä kerroksena toimii esityskerros, jossa kaikki tiedon esitys tapahtuu (lukuun ottamatta vahvistuksia ja virheilmoituksia). Täysin puhtaasta esityskerroksesta ei ole kyse, koska sitä kontrolloivaa kerrosta ei ole toteutettu.

Järjestelmän kerrosarkkitehtuuri muodostuu kuudesta kerroksesta. Näiden kerrosten looginen suhde on esitetty alla (kuva 2.4.). Kuvasta voidaan havaita, että palvelukerros hyödyntää suoraan käyttöjärjestelmäkerrosta. Toiminta ei ole yleistä ohjelmassa, mutta sitä yhteyttä on hyödynnetty tehokkuuden ja joidenkin käyttöjärjestelmän viestien hyödyntämiseen. Toinen huomionarvoinen piirre mallista on kerroksien 3-6 ja tietokantakerrosten välissä oleva suora yhteys.

Kuva 2.4. *Todellinen kerrosarkkitehtuuri.*

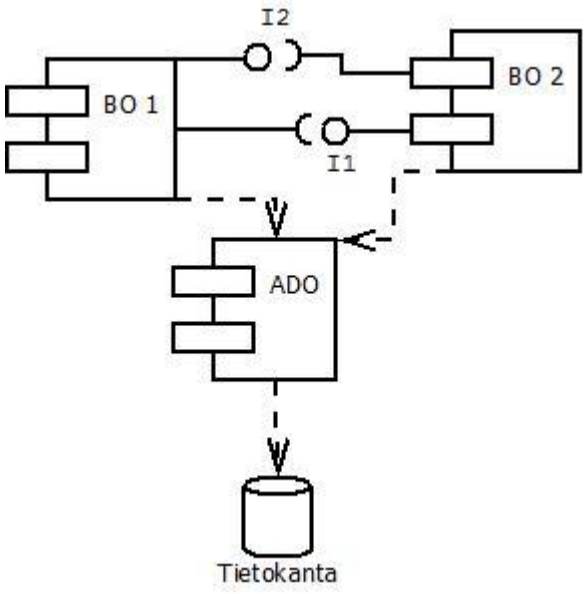
2.2.2. Tietovarastot

Tietovarastoarkkitehtuurissa järjestelmä ajatellaan rinnakkain toimivina alijärjestelminä, jotka jakavat tietoa yhteisen alustan kautta (*repository*). Usein tietovarasto on tietokanta, vaikka muitakin vaihtoehtoja on. Arkkitehtuuri muistuttaa hyvin paljon asiakas-palvelin (*client-server*) mallia. Huomattavimpana erona näillä kahdella on se, että asiakas-palvelin-mallissa asiakkaan roolissa toimiva osa ei ole riippuvainen tietovaraston toimintatavasta. Toinen ero on siinä, että tietovarasto-mallissa tietovaraston käyttäjä sisältää enemmän toiminnallisuutta, asiakas-palvelin-mallissa toiminnallisuus painottuu enemmän palvelimen puolelle. Kuvassa 2.5, on esitetty yleistys tietovarastosta ja sen suhteesta alajärjestelmiin.

Kuva 2.5 *Tietovarastoarkkitehtuurin yleistys.*

Kuva 2.5 esittää tilannetta, jossa alijärjestelmä 1 ja alijärjestelmä 2 kommunikoivat tietovaraston kautta. Tietovarastomallilla toteutettujen mallien heikkous tulee esille rinnakkaisessa käytössä silloin, kun esimerkiksi tietovarasto on lukittuna. Tavallisessa käytössä tietovaraston lukkiutumista, tiedon hakemista tai kirjoittamista varten ei välttämättä edes huomaa. Vikatilanteessa alijärjestelmä ei välttämättä avaa lukitusta, jolloin muiden pyynnöt tietovarastoon estyvät.

Järjestelmässä kaikki keskeinen tieto on tallennettuna yhteen keskeiseen tietokantaan. Kyseessä olevan ohjelmiston tietokanta voi olla *Access*-kanta tai *Microsoftin SQL Server*. Kummassakaan tapauksessa tietokannan sijainti ei ole rajoitettu samaan koneeseen kuin sitä käyttävä ohjelma. Access-tietokannassa useamman yhtäaikaisen käyttäjäsession kanssa voi kuitenkin tulla ongelmia. SQL Serveriä käytettäessä myös hajautus tulee mahdolliseksi. Tietovuoarkkitehtuurissa on tarkoitus esittää tietovarastoon, tietokantaan, liittyvät komponentit. Käytettäessä ADO:a voidaan yksinkertaistaa tilanne alla olevan mukaiseksi (kuva 2.6.). Kuvan oleellisimmat tiedot ovat se miten tietoa noudetaan yhteisestä tietovarastosta, sekä se miten tilatietoa ylläpidetään eri komponenttien välillä. ADO-kerros huolehtii sitä, että käytettävä tieto on relevanttia ja liiketoimintayksiköt (BO 1 ja BO 2) kommunikoivat keskenään. Tarkoituksenmukaista ei ole välittää olion tietoja liitäntöjen kautta, koska kyseinen data voidaan noutaa suoraan tietovarastosta. Menetelmä on erityisen tehokas silloin kun haluttu tieto ei ole samassa ohjelmakirjastossa.

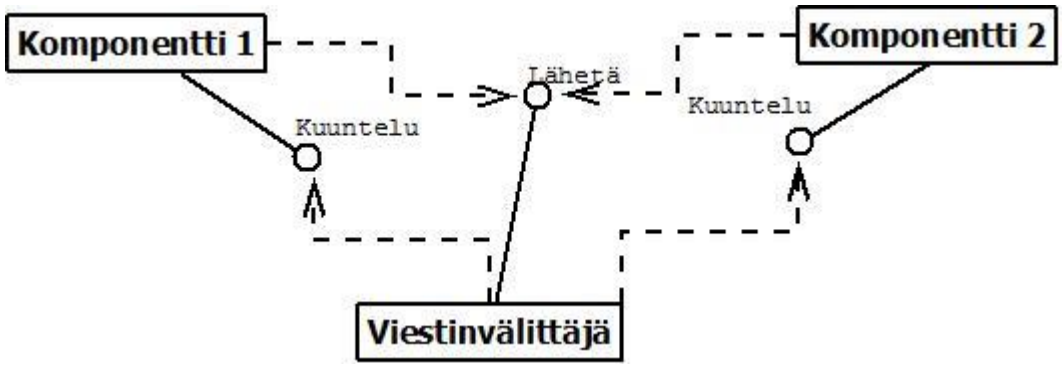


Kuva 2.6. *Tietovuon rakenne.*

Järjestelmän toteuttaminen käyttäen tietovarastoarkkitehtuuria vaatii kehittäjiltä johdonmukaisuutta. Tilanteissa, joissa käytetään transaktioita, täytyy varmistua transaktion päättämisestä. Usein tietokannoissa voi erikseen määrätä katkaisuajan, jonka jälkeen transaktio lopetetaan ja muutokset perutaan. Ympäristöissä, joissa käyttäjävolyymi on suuri, saattaa ilmetä paljon lukituksia. Ongelmaa on minimoitu lukitsemalla vain tarvittavat rivit ohjelmassa ja sitomalla transaktiot käyttäjäkohtaisiin sessioihin. Toinen suunnittelun kannalta merkittävä päätös on tehtävä, kun mietitään, miltä tasolta ADO-kerrosta on hyvä kutsua. Kuva 2.6. esittää tilannetta, jossa liiketoimintakerros hoitaa yhteyksiä ADO:n kanssa. Yleisempänä toimintatapana on transaktioiden hoitaminen palvelukerrokselta, jolloin liiketoimintakerrokselle jää ainoastaan yksityiskohtaisempi tietokantaoperaatioiden suorittaminen. [5, pp. 649-650]

2.2.3. Viestinvälitysarkkitehtuuri

Viestinvälitysarkkitehtuurissa (kuva 2.7.) jokainen komponentti toteuttaa saman rajapinnan viestien vastaanottamiseen ja rekisteröityvät kuuntelijoiksi [2, p. 258]. Arkkitehtuuri mahdollistaa sen, että myös komponentti voi lähettää viestejä viestinvälittäjälle. Tätä toimintamallia nimitetään lähetykseksi (*Broadcast*) [2, p. 258]. Toinen saman arkkitehtuurityylin alle luettavaksi toimintamalliksi lasketaan keskeytyksiin perustuva toiminta, jota hyödynnetään usein IO-toiminnoissa.

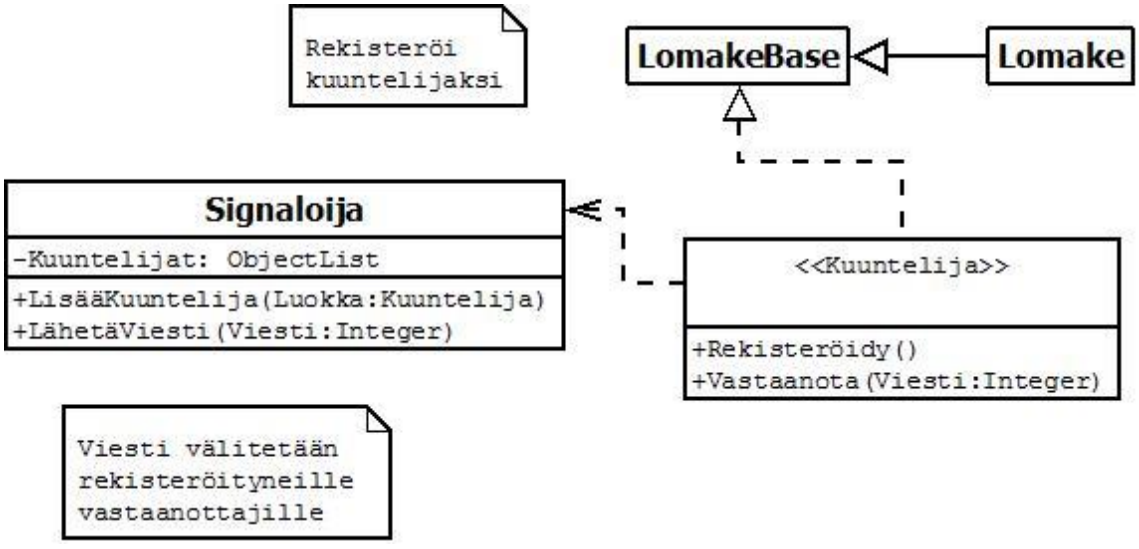


Kuva 2.7 *Tiedonvälitysarkkitehtuurin toimintaperiaate.*

Viestinvälitysarkkitehtuurissa heikkoutena on sen aiheuttamien viestien määrä. Suuri viestien määrää heikentää järjestelmän suorituskykyä. Toinen arkkitehtuurityylin huomattava heikkous on se, että niitä toteutettaessa tulisi ymmärtää myös se, mihin tarkoitukseen ne on suunniteltu. Huonolla toteutuksella voi olla vaikutuksia koko järjestelmän toimintaan.

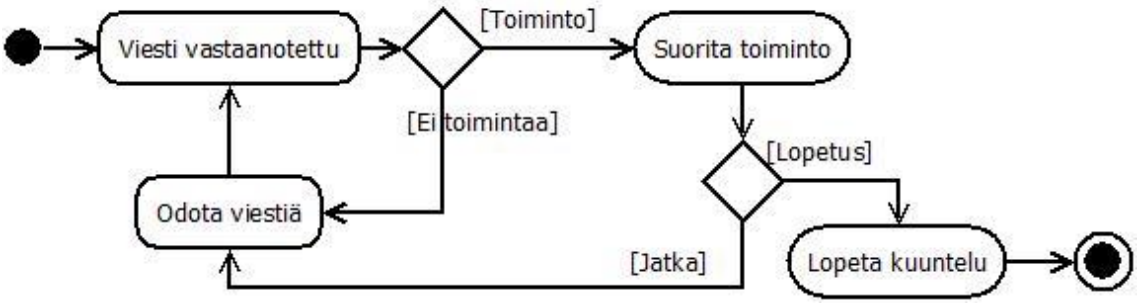
Viestinvälitysarkkitehtuuri on parhaimmillaan tilanteissa, joissa viesti voidaan esittää lyhyesti, eikä siihen vaadita vastausta. Ohjelmassa viestinvälitysarkkitehtuuria toteutetaan informoimaan ohjelman sulkemisesta sekä lisenssin mukaisista käytännöistä.

Arkkitehtuurin periaatteiden mukaisesti komponentit rekisteröityvät signaalin lähettäjälle, jotta voivat vastaanottaa sanomia. Toteutuksena idea on yksinkertainen. Jokainen ohjelman ikkuna on peritty luokasta, joka toteuttaa kuuntelurajapinnan. Ikkunaa luotaessa se rekisteröityy kuuntelemaan signaaleja (kuva 2.8.).



Kuva 2.8. *Yleistys viestinvälityksen toteutuksesta.*

Viestit eivät ole käskyjä vaan ilmoituksia. Niiden perusteella kukin ikkuna voi toimia haluamallaan tavalla (kuva 2.9.).



Kuva 2.9. *Viestinvälityksen toiminta ohjelman ikkunan ja ytimen välillä.*

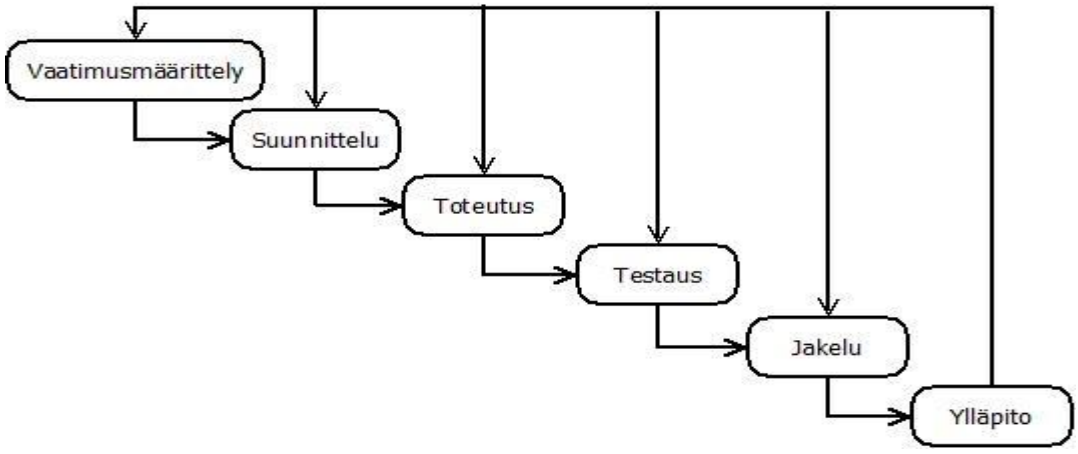
Esimerkkinä on ikkuna, jolle tulee sammutuspyynnön sisältävä signaali. Se voi joko kehottaa käyttäjää tallentamaan muutokset tai automaattisesti huolehtia, että tietoa ei katoa.

3 EVOLUUTION HALLINTA

Ohjelmistotuotteen kehityksen elinkaari alkaa ensimmäisestä määrittelystä ja päättyy viimeiseen muutokseen. Tuotteen käyttö saattaa jatkua huomattavasti pidemmälle, mutta kehityksen osalta toiminta on päättynyt. Näiden kahden ajanhetken välillä ohjelmaa mukautetaan, laajennetaan ja parannetaan. Ajan kuluessa tehdyistä muutoksista muodostuu monimuotoinen kirjo lähdekoodiin. Muutokset kuvastavat hetkellisiä tarpeita, hyödynnettyjä tekniikoita ja käytettyjä alustoja.

3.1 Prosessimallit

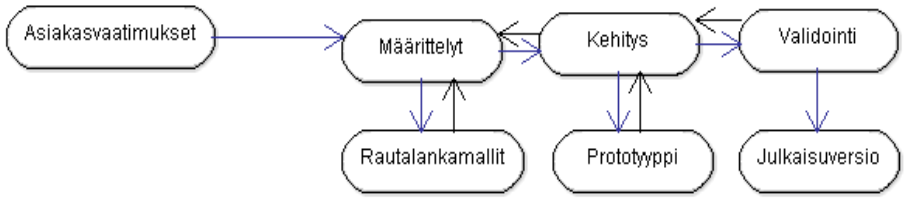
Ohjelmistotuotannon ensimmäinen varsinainen prosessimalli on johdettu yleisestä järjestelmäprosessista [2, p. 66] . Se on luonteeltaan hyvin muodollinen ja tarkasti määritelty prosessi. Siinä vaiheet kuljetaan tietyssä järjestyksessä ilman poikkeuksia. Se tunnetaan paremmin vesiputousmallina (kuva 3.1.). Jokainen vaihe voidaan aloittaa vasta, kun aiempi on saatu valmiiksi. Normaalisti vesiputousmallissa tehdään vain muutama iteraatio, koska ne vaativat hyvin paljon työtä ja tästä syystä ovat myös kalliita kustannuksellisesti ja ajallisesti. Mallia voi nykyisissä ohjelmistoprojekteissa hyödyntää silloin, jos asiakasvaatimukset eivät muutu (ts. todennäköisyys on hyvin pieni) tai se on osa suurempaa projektia. [2, p. 67]



Kuva 3.1. *Vesiputousmalli.*

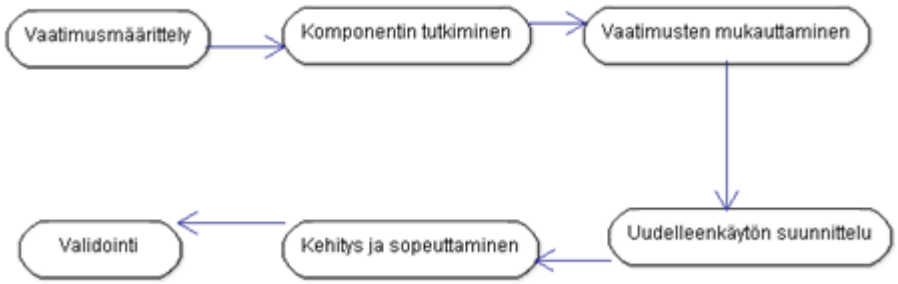
Vesiputousmallille on syntynyt vastapainoksi evoluutiomalli (kuva 3.2.). Sen tarkoituksena on mahdollistaa ohjelmiston toteuttaminen muuttuvien määritysten mukaan. Malli soveltuu hyvin pieniin projekteihin ja pienille kehitystiimeille, koska se on hyvin kevyt dokumentoinnin osalta. Jokaisesta mallin iteraatiosta ei ole mielekästä tehdä uutta määrittelyä ja suunnittelua. Pitkän elinkaaren tuotteisiin malli ei sovellu hyvin,

koska jokainen muutos voi vaikuttaa sovelluksen arkkitehtuuriin hyvin paljon. [2, pp. 68-69]



Kuva 3.2. *Evoluutiomalli.*

Pitkän elinkaaren tuotteeseen parhaiten soveltuu komponentti-pohjainen malli (*Component-based software engineering CBSE*). Siinä pyritään hyödyntämään aiemmin luotuja komponentteja joko suoraan tai käärimällä (*wrapping*). Hyödynnettävät komponentit voivat olla yksittäisiä olioluokkia tai valmiita ostettuja laajoja kokonaisuuksia (*Commercial off-the-shelf systems COTS*). Komponenttien käyttö edellyttää niiden toiminnan tarkoituksen tuntemista, jotta voidaan määritellä toimintaympäristö yhteensopivaksi ja mahdollisesti tehdä tarvittavat muutokset myös itse hyödynnettävään komponenttiin (kuva 3.3.). Mallin haittapuolena on mahdollinen suorituskyvyn heikkeneminen, koska yleiseen käyttöön tarkoitettujen komponenttien tulee suoriutua monenlaisista tilanteista. [2, pp. 440-442]



Kuva 3.3. *Komponenttipohjainen prosessimalli [2, p. 70].*

CBSE-mallissa järjestelmä ja komponentti mukautuvat toisiinsa. Usein mukautuminen toteutetaan järjestelmän puolelle, koska komponentti halutaan pitää toimivana tai koska sitä ei ole mahdollisuutta muuttaa (esimerkiksi osaa COTS:sta ei ole mahdollista muokata).

3.2 Ylläpidettävyys

Ohjelmiston ylläpidettävyys kuvaa ylläpitotoimenpiteiden vaativuutta. Pitkän elinkaaren ohjelmistotuotteessa ylläpidettävyys laskee ajan kuluessa. Syynä heikkenemiseen on ohjelmiston lisääntyvä monimutkaisuus [1, p. 66]. Huono ylläpidettävyys saattaa johtaa

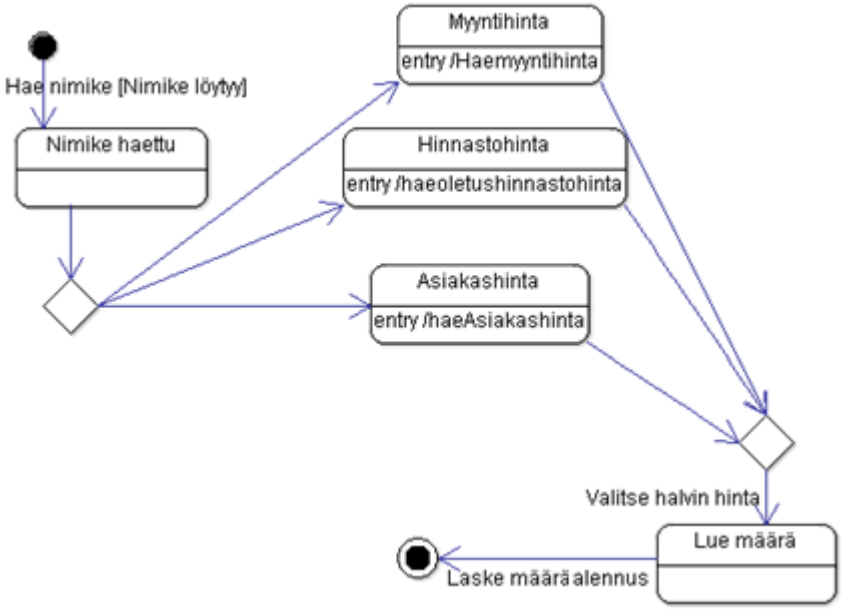
vajavaisiin ratkaisuihin nopeiden tulosten saavuttamiseksi. Siitä saattaa syntyä kierre, jossa heikennetään entisestään ylläpidettävyyttä. [1, p. 18]

Evoluutiolaki ylläpidettävyyden heikkenemisestä muutoksista johtuen selittää asian osittain. Muitakin vaikuttavia tekijöitä löytyy, kuten huonot suunnittelu- ja toteutusratkaisut, yleisen ohjeistuksen puuttuminen tietomäärittelystä ja dokumenttien riittämättömyys [1, p. 58, 6]. Jokaisen edellä mainitun osan merkitykseen on perehdyttävä pitäen mielessä toimiala ja kehittäjien vähäinen määrä.

Huonot suunnittelu- ja toteutusratkaisut voivat johtua tarpeiden alimitoittamisesta alustavassa toteutusvaiheessa. Ongelmana pitkään käytössä olevassa ohjelmassa on se, että vuosia sitten suunnitellut käyttötarpeet eivät välttämättä pysy muuttumattomina. Ylläpidettävyyden ja jatkokehittämisen kannalta modulaarisuuden lisääminen on kannattavaa. Vahvasti modulaarista ohjelmaa voidaan hyödyntää samoin kuin CBSE-mallissa (*Component Based Software Evolution*) komponentteja. Menetelmän hyödyntäminen vaatii kuitenkin ymmärryksen siitä, milloin vanha komponentti on korvattava uudella. Pitkällä aikavälillä tapahtunut komponentin huonontuminen on vaikea havaita erikseen tarkastelemalla. Usein ongelma havaitaan tehtäessä korjausta tai lisättäessä muutosta, jolloin joudutaan pohtimaan nykyisen toteutuksen luonnetta. Havaittaessa ongelman kehittäjän tulisi arvioida tarkemmin komponentin ajantasaisuutta. Todetessaan muutostarpeen tulee se kirjata tutkittavana muutospyyntönä, jolloin ryhmän kesken voidaan arvioida mahdollisia vaikutuksia ja aikataulua. Tällaisissa tilanteissa on tärkeätä, että ryhmän jokainen jäsen osallistuu arviointiin. Siten saadaan arvioitua kattavammin komponentin vaikutusaluetta ja erilaiset hyödyntämisnäkökohdat. Itsearviointia tutkitaan tarkemmin Jatkokehittäminen-kappaleessa (3.3).

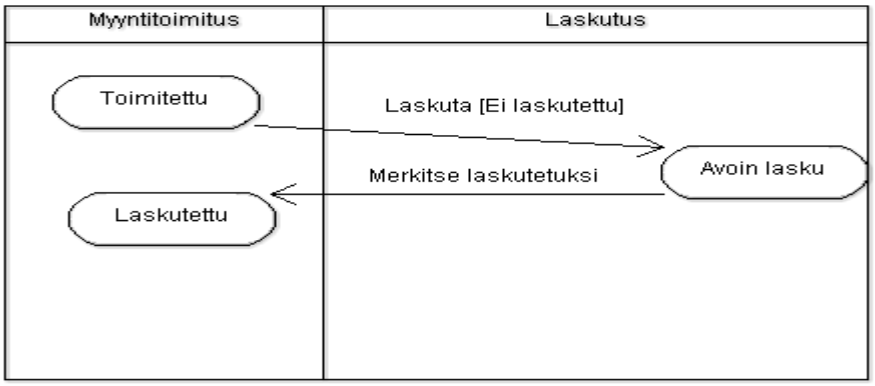
Pienessä kehitysyhteisössä tarve järjestelmän sisäiselle dokumentaatiolle on melko pieni, koska kehittäjät voivat olla paljon toistensa kanssa vuorovaikutuksessa. Ajan kuluessa asioita kuitenkin unohtuu ja ryhmän henkilöt vaihtuvat. Siitä johtuen dokumentaatiota täytyy luoda ylläpidettävyyden säilyttämiseksi.

Ohjelmiston tietomäärittely on tärkeä osa ylläpidettävyyttä, koska ilman suunnitelmallisuutta tietovirran kulussa on vaikeampaa havaita poikkeamia. Taloushallinnon ohjelmassa osa tietomäärittelystä on dokumentoitu laissa. Lain tai asetuksen muuttuessa ohjeistamattomista tietomääreistä ei pysty nopeasti arvioimaan tarvitaanko muutoksia muualle ohjelmaan, jotta se täyttäisi vaatimukset. Ne ohjelman osat, joissa tiedosta ei ole asetettu lailla, täytyisi myös sisällyttää tietomäärittelyihin. Tilanteissa, joissa ohjelman ulosanti ei ole täysin selvä tai se muodostuu monesta eri vaiheesta, voidaan esittää sitä tilakaavion avulla. Kuva 3.4. esittää ratkaisua ohjelman osien tietomäärittelyä tilakaavion avulla nimikkeen laskutettavan hinnan määrittelemiseksi. Esimerkissä halutaan havainnollistaa erilaisten ulosantien mahdollisuutta.



Kuva 3.4. *Esimerkki: nimikkeen hinta.*

Usein lähdekoodia tutkimalla (esim. takaisinmallintamalla) saadaan toiminnallisuus selville, mutta joissakin tilanteissa voi olla hankalaa ymmärtää joidenkin toimintojen tarkoitusta tai hahmottaa epäsuoria riippuvuuksia ohjelman sisällä. Pienellä kehittäjämäärällä dokumentoinnin lisääminen on suuri haaste, koska se lisää paljon työtä. Toteutettaessa kokonaan uusia osioita ohjelmaan, dokumentaation luonti on suhteellisen helppoa. Edellytyksenä helppoudelle on se, että ominaisuudet toteutetaan dokumentaation pohjalta ja niitä ylläpidetään säännöllisesti. Tällöin työmäärä lisääntyy hieman, mutta ei aiheuta suuria toimenpiteitä. Olemassa olevien osioiden täydellinen takaisinmallintaminen määrittelyiksi ei ole pakollista, koska niissä muutokset ovat lähinnä korjauksia. Tärkeimmät niistä luotavat dokumentaatiot ovat tietomäärittelyijä (esim. kuvan 3.4. kaltainen) ja aktiviteetti-kaaviot (esim. kuva 3.5.). Alla on esitetty aktiviteettikaaviossa yksinkertaistus tarpeenmukaisesta dokumentoinnista aktiviteetti-kaavion avulla. Esimerkki-kuvassa (kuva 3.5.) on esitetty yksinkertaistus myyntitoimituksen laskutuksesta.



Kuva 3.5. *Aktiviteetti-kaavio myyntitoimituksen laskutuksesta.*

Aktiviteettikaavion tarkoituksena on luoda kokonaiskuva jostakin ohjelman toimintojen keskinäisistä riippuvuussuhteista. Siinä ei ole tarkoituksenmukaista esittää yksityiskohtaisesti jokaista ohjelman metodia, vaan tuoda esille toiminnon oleellisimmat piirteet ja toteutusjärjestys. Sekvenssi-diagrammilla voidaan tarpeen mukaan tarkentaa epäselviä yhteyskohtia.

3.3 Jatkokehittäminen

Ohjelmisto on työkalu, jota kehitetään sen ollessa aktiivisessa käytössä. Kohdassa Ylläpidettävyys (3.2) käsitelty ylläpito on luonteeltaan korjaavaa kehitystä, kun taas jatkokehittämisen tarkoituksena on luoda lisäarvoa loppukäyttäjälle. Lisäarvoa saadaan tehtyä toimintoja parantelemalla sekä lisäämällä ominaisuuksia. Jatkokehittäminen voi olla luonteeltaan pienimuotoista, ohjelman ylempiin tasoihin vaikuttavaa (esim. tietokentän lisääminen lomakkeelle) tai erittäin laajamittaista, sisältäen joka ohjelmakerrokseen vaikuttavia muutoksia. Kokonaisvaltaisemman jatkokehityksen kohdalla voidaan tietyissä tapauksissa puhua ohjelman uudistamisesta (*re*-engineering). Luku Uudistaminen (3.4) käsittelee ohjelmiston kokonaisvaltaista uudistamista; sellaista, jossa se kirjoitetaan osittain tai kokonaan uudestaan.

Tässä kohdassa perehdytään jatkokehittämiseen osana ohjelman evoluutiota. Osana tutkimusta tarkastellaan jatkokehittämisen haasteita ja riskejä. Pyrkimyksenä on tunnistaa erilaisia muutostyyppejä jatkokehityksessä. Tarkastelun tuloksena pyritään luomaan muutoksista aiheutuvien ongelmien ehkäiseviä toimintamalleja. Tarkoituksena ei ole luoda kaikenkattavaa muutos-riski-ratkaisu-tutkimusta. Haastetta lähestytään sen loogisessa toteutusjärjestyksessä, määrittelyssä aikataulun kautta toteutukseen ja siitä päivityksen luomiseen.

Jatkokehittämisprojektit voivat olla laajuudeltaan hyvin erilaisia, ilmoitusten ja dialogien lisäämisestä kokonaan uuden osion luomiseen. Pienimpiä projekteja lukuun ottamatta muutosvaikutusanalyysin luominen on kannattavaa [1, p. 84]. Se on dokumentti, jossa pyritään arvioimaan muutoksen vaikutuksia ohjelmaan. Pienellä kehittäjätiimillä muodolliset ja kaikenkattavat muutospyyntödokumentit eivät ole ajankäytöllisesti järkeviä. Parempi tapa on muodostaa ratkaisun pääkohdista yhteenveto keskustelemalla joko koko tiimin tai joidenkin sen jäsenten kesken. Yhtenä osana sitä määritellään uudet ja lisätyt tietokentät. Niiden huomioiminen erikseen on tärkeätä, koska silloin voidaan suunnitella päivitystoimenpiteitä erillään ja ennen ominaisuuden valmistumista. Kavennettuun muutospyyntödokumenttiin voidaan lisätä myös tiedostetut riskit ratkaisussa, sillä niiden perusteella voidaan kohdistaa testausta epävarmempiin osiin sekä määritellä suuntaviivat jatkokehitykselle. Tunnistettujen riskien perusteella testauksessa voidaan käyttää aikaa enemmän riskikohtien testaamiseen.

Muutosvaikutusanalyysin jälkeen on vuorossa projektin aikataulun luonti. Ajankäytön tärkeimpänä arviointikohteena on se, saadaanko lopullinen tuotos valmiiksi siten että se ei pidennä tulevan version julkaisua. Tilanteissa, joissa vaikuttaa siltä, että ominaisuuden luominen ja testaaminen vievät enemmän kuin yhden version, täytyy sitä

kehittää erillään muusta ohjelmasta. Haittapuolena erillään kehittämisessä on muiden muutosten oleminen irrallaan, jolloin joitakin asioita voidaan joutua testaamaan uudelleen. Käytettäessä muutossetteihin (*changeset*) perustuvia versiohallintajärjestelmiä (esim. mercurial ja git), uudet ominaisuudet voidaan tuoda mukaan sitä mukaa, kun ne lisätään varsinaiseen versioon.

Ohjelmistoa samanaikaisesti laajennettaessa sekä ohjelmavirheitä korjattaessa voi tulla vastaan tilanteita, jolloin useampi kehittäjä käsittelee rinnakkain samaa toiminnallisuutta. Tilanteessa, jossa korjaus lisätään ja testataan ennen uutta ominaisuutta, voi ilmetä ongelmia. Esimerkissä nimikkeen varastomuutoksesta (esimerkki 3.1.) on esitelty yksi mahdollinen tapa, miten sellainen voi muodostua. Hyvällä testisuunnitelmalla voidaan havaita, että muuttunut toiminnallisuus ei ole siirtynyt mukana uuteen ominaisuuteen. Kattavan testauksen lisäksi voidaan hyödyntää ristikorrelaatioon perustuvia toimintatapoja [7, p. 214]. Sen tarkoituksena on, että samaan alueeseen tehtyjä muutoksia verrataan keskenään. Vaatimuksena on, että tiimin sisällä on tieto siitä, mitä milloinkin ollaan tekemässä, jotta on mahdollista havaita korreloivia muutoksia. Myös tästä syystä on hyödyllistä, jos koko tiimi osallistuu ominaisuuden muutosvaikutusanalyysin luomiseen. Käytännössä tekijöiden on keskenään neuvoteltava muutoksista niiden valmistuttua. Tämän lisäksi voidaan hyödyntää työkaluja, kuten Kiln:ä ja lähettää sieltä koodikatselmointipyyntö toiselle osapuolelle, jolloin lähettäjän toteutus on heti nähtävillä. Rinnakkain kehittämisessä on silti haasteensa, vaikka päällekkäisiä muutoksia tutkittaisiin huolellisesti.

```
      Lähtötilanteessa jokaiseen nimikkeeseen liittyy tietty varasto.
Toiminnassa on havaittu ongelma, kun nimikkeelle ei ole määritelty
varastoa.  Samalla halutaan, että nimike voi liittyä useampaan kuin
yhteen varastoon. Nimikkeelle halutaan jättää varastotieto kuvaamaan
oletusvarastoa. Ensin toteutetaan korjaus, jossa varmistetaan että
nimikkeelle asetetaan jokin varastotieto. Korjaus testataan ja
todetaan toimivaksi. Samalla on kehitetty usean varaston ominaisuutta,
jossa erilliseen tietokantatauluun tehdään n-n liitos nimikkeiden ja
varastojen välille. Ominaisuutta testataan ja sen todetaan toimivan.
Kuitenkaan aiemmin tehty korjaus ei luo pakollista varastoa uuteen
tietokanta-tauluun ja näin virhe siirtyy uuteen ominaisuuteen.
```

Esimerkki 3.1. *Nimikkeen varastomuutos.*

Luotaessa toiminnallisuutta, joka vaatii muutoksia tallennettavan tiedon käsittelyyn, täytyy huomioida myös ohjelmiston päivittäminen asiakaspäässä. Päivitystä suunniteltaessa voidaan hyödyntää muutosvaikutusanalyyseihin kerättyjä tietojen muutoksia. Muutokset voidaan jaotella kolmeen kategoriaan vaadittavien toimenpiteiden perusteella: 1) Tiedon suora muuntaminen (kuva 3.6.) 2) Viitetiedon muokkaaminen (kuva 3.7.) 3) Tietojen yhdistäminen (kuva 3.8.).

Tiedon suorassa muuntamisessa tietokentissä olevat arvot päivitetään ennalta määriteltyjen sääntöjen mukaisesti uusiin arvoihin. Päivitystä luotaessa arvioidaan kaikki mahdolliset nykyiset kentän arvot ja annetaan niille uusi vastinpari, johon ne muun-

netaan. Tuntemattomissa tilanteissa kentälle annetaan erikoisarvo, jota voidaan päivityksen jälkeen tutkia paremmin.

Lähtötilanne

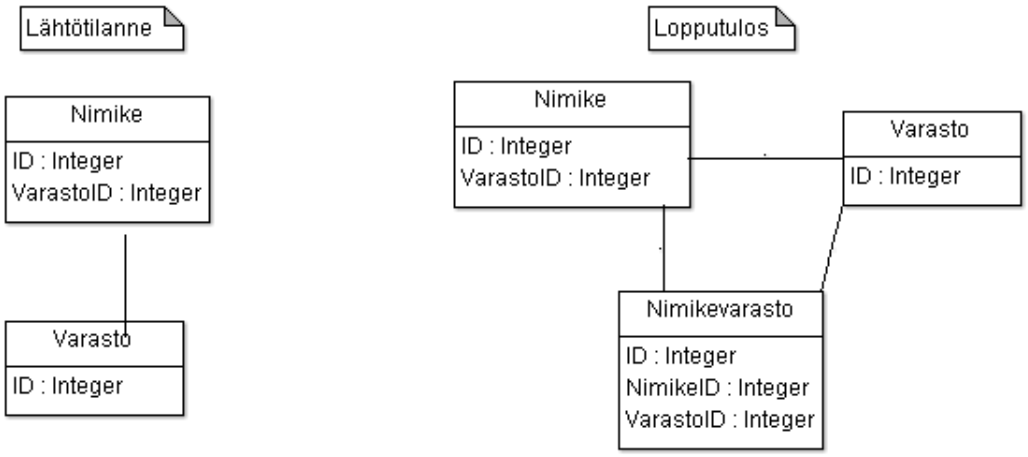
ID	Tila
1	2
2	1
3	3
4	4
5	1
6	2

Lopputulos

ID	Tila
1	1
2	1
3	3
4	4
5	1
6	1

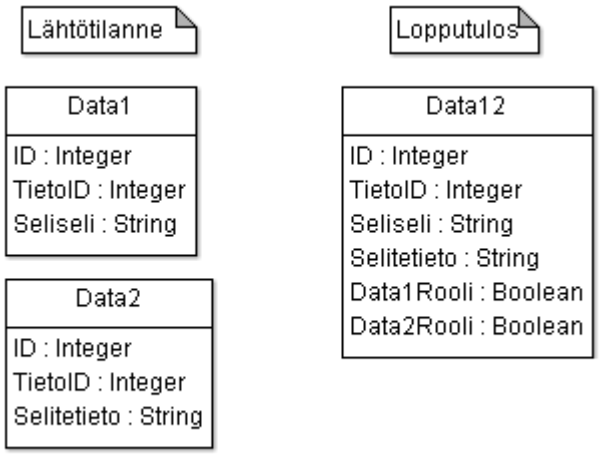
Kuva 3.6. *Tiedon suora muuntaminen.*

Viitetietoa muutettaessa tieto on olemassa, mutta sen sijainti täytyy muuttaa. Esimerkki 3.1 kuvaa tilannetta, jossa viitetieto muuttuu. Viitetiedon muutoksen takia päivityksessä luodaan aiempien tietojen perusteella uudet tiedot viitaten alkuperäisiin kohteisiin. Mahdollisina poikkeuksina ovat tyhjät (null) arvot. Tilanteesta riippuen ne voidaan joko jättää huomioimatta tai alustaa jollakin arvolla. Päivityksen luonnissa niihin on kohdistettava erityishuomiota, jotta viite-eheys säilyy.



Kuva 3.7. *Viitetiedon muokkaaminen.*

Kolmannessa kategoriassa tyypillinen tilanne on, että ohjelma sisältää samankaltaista tietoa, jota halutaan yhdistää eri lähteistä. Kuvan 3.8. tilanteessa taulut *Data1* ja *Data2* on yhdistetty *Data12*-tauluksi. Kummassakin alkuperäisessä taulussa on *TietoID*-kenttä, jonka avulla tiedot liitetään. Todellisessa tilanteessa kyseessä voisi olla kaksi henkilötietoja sisältävää taulua, joissa yhdistävänä ominaisuutena olisi henkilötunnus.



Kuva 3.8. *Tietojen yhdistäminen.*

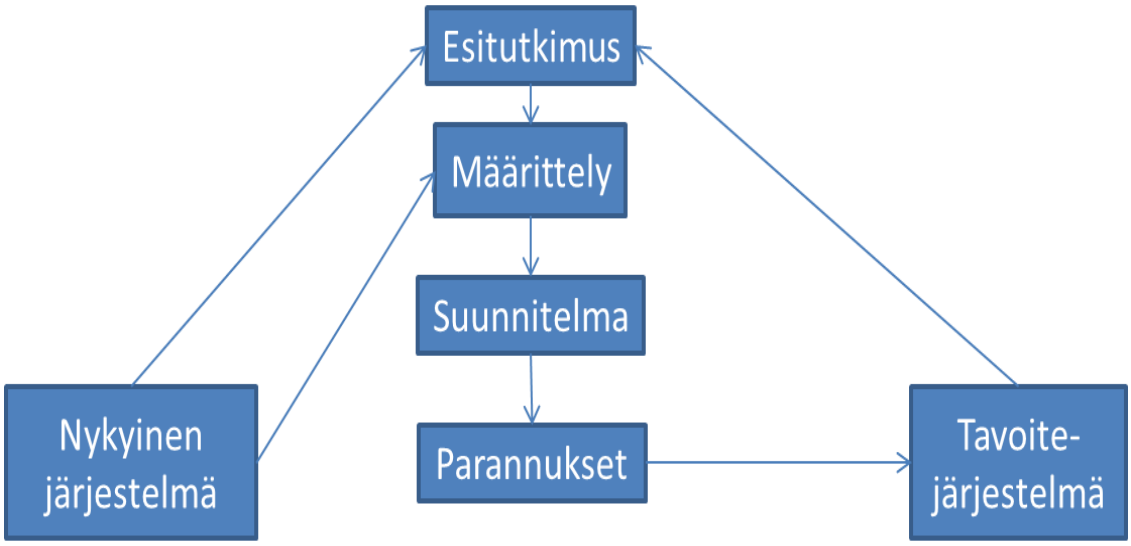
Käytännössä aina ei kuitenkaan ole yhtä yksilöivää tietoa tai tieto voi löytyä useaan kertaan. Tällaisessa tilanteessa voi olla tarpeen antaa käyttäjän tehdä päätöksiä, jotta ohjelman julkaisu ei viivästy turhan monipuolisesta päivityksestä ja sen toimivuuden testauksesta. Mahdollistamalla käyttäjän syötteet virheiden todennäköisyys kasvaa huomattavasti. Käyttäjän tekemiin virheisiin voi varautua tekemällä mahdollisuuden peruuttaa muutokset päivityksen jälkeen.

3.4 Uudistaminen

Tuotteen elinkaaren aikana saattaa tulla vastaan tilanne, jolloin toteutetaan ohjelmiston uudistaminen (*re-engineering*). Sen tarkoituksena on ohjelman toteuttaminen uudelleen käyttäen ajantasaisia ja hyväksi havaittuja menetelmiä toteutuksessa. Uudistaminen voi olla ennakoiva toimenpide, jolloin sillä pyritään varautumaan tulevaisuudessa haasteisiin. Syy uudistamistarpeelle voi tulla vastaan myös ohjelmaa laajennettaessa, jolloin joidenkin ominaisuuksien lisääminen ei välttämättä ole mahdollista ympäristön rajoitteiden tai arkkitehtuuristen ratkaisujen vuoksi mahdollista. Suuremmissa muutoksissa, kuten käyttöympäristöä tai ohjelmointikieltä vaihdettaessa, tarvitaan laaja-alaista uudistamista, jossa toimenpiteitä ei suoriteta yhdessä osassa, vaan monessa peräkkäisessä lisäyksessä [2, p. 501].

Uudistaminen muistuttaa tietyiltä osin jatkokehitysprojektia, mutta sisältää myös selviä eroavaisuuksia. Sen tärkeimpänä osana on olemassa olevan järjestelmän ymmärtäminen. Se voidaan saavuttaa tutkimalla aiemmin luotuja dokumentteja (määrittelyt, muutosvaikutusanalyysit yms.) tai takaisinmallintamalla (*reverse engineering*) dokumentaation ollessa vajavaista. Havaintoja voidaan muodostaa niiden lisäksi ohjelman suorituksen aikaista toimintaa tutkimalla. Takaisinmallinnuksen ja aiemman dokumentaation tueksi suoritetaan esitutkimus, joka kartoittaa tarkemmin itse muutostyötä. Kuvassa 3.9. esitetään uudistamisprojektin toteutuksen osiot sekä niiden välillä olevat yhteydet. Kuvan tärkeimpänä viestinä on se, että tavoitejärjestelmää voidaan jalostaa uudelleen esitutkimuksen kautta. Syy siihen, että tavoitejärjestelmää ei lähdetä paranta-

maan uudelleen suoraan määrittelystä tai suunnittelusta on se, että silloin voisi jäädä jokin alkuperäisen ohjelman piirre huomioimatta.



Kuva 3.9. *Tavanomaisen uudistamisen rakenne.*

Uudistamisprojektia aloitettaessa on oltava selkeä tavoite siitä, millaista hyötyä muutoksilla haetaan, millaisia riskejä se sisältää liiketoiminnallisesti sekä kuinka kauan sen toteuttaminen kestää [1, p. 166]. Tässä tutkitaan kahta uudistamistyyppiä: vähittäistä ja yhtäkkistä. Vähittäinen on komponenteittain tapahtuvaa muunnosta, josta voidaan julkaista välissä asiakkaalle näkyviä versioita. Yhtäkkisessä uudistamisessa käyttäjä siirtyy yhdellä kertaa kokonaan uudistettuun ohjelmaan

Uudistaminen voidaan jaotella kahteen projektivaiheeseen: esitutkimukseen (kuva 3.9. *Esitutkimus*) ja muutosvaiheeseen (kuva 3.9. *Määrittely*, *Suunnitelma* ja *Parannukset*). Esitutkimuksen tarkoituksena on olla apuna päätöksenteossa toteutukseen ryhtymisestä vastaavasti kuin jatkokehityksessä muutosvaikutusanalyysilla tilannetta kartoitettaessa. Muutosvaihe-osuus muistuttaa hyvin paljon normaalin projektin toteutusta, mutta lopputulokselle on olemassa valmis vertailukohta.

Prosessi uudistamisen läpiviemiseksi, ennen esitutkimusta, alkaa aina tarpeen tunnistamisesta. Tarve voi syntyä ohjelmistoalustan ylläpidon päättymisestä, liian paljon aikaa vievästä ylläpidosta tai teknisen edun tavoittelusta. Edellä mainitut tarpeet ovat esimerkkejä mahdollisista syistä, mutta eivät ainoita mahdollisia.

Esitutkimuksen tarkoituksena on luoda selvä näkemys tarvittavista toimenpiteistä, kustannuksista sekä aikataulusta. Tarkimman arvion kestosta saa tutkimalla alkuperäiseen toteutukseen kulunutta aikaa, jos sellaista tietoa on saatavilla. Tilanteessa, jossa tietoa luotettavan aikataulun luomiseksi ei ole, täytyy sellainen muodostaa ryhmän kesken. Arvio on sitä tarkempi, mitä pienempiin osiin aikataulu voidaan jakaa [2, p. 99]. Arviota ositettaessa tulee kuitenkin pitää osat järkevän mittaisina, jotta kestojen pyöristyksistä ei aiheudu itsessään suurta virhettä. Esimerkiksi noin vuoden kestävän projektin osien arvioinnissa päivätasoinen aikataulutus on todennäköisesti jo harhaanjohtavaa,

mutta viikkotasoisella arvioinnilla päästään lähelle totuutta kokonaisuudessa. Pahimmillaan kehitysympäristön muuttuessa toteutus voi vaatia uudenlaisia menetelmiä, jotka ovat ennestään tuntemattomia. Silloin arvion tekeminen etukäteen on lähes mahdotonta. Vasta työn edetessä pystytään muodostamaan arvioita kestosta.

Aikataulutusta seuraa henkilöresurssien jakaminen. Pienellä kehittäjämäärällä henkilöresurssien jakaminen paljon aikaa vievälle projektille on hoidettava mahdollisimman tehokkaasti. Uudistamisprojektiin kiinnitetyt henkilöt vähentävät uusien ominaisuuksien lisäyksiä ja vähemmän kriittisten virheiden korjaaminen pitkittyy. Yksi mahdollisuus resurssien jakamiseen on hetkittäin siirtää uudistamisprojektin tekijät nykyohjelmaan toteuttamaan uusia ominaisuuksia ja sitten palata takaisin uudistamisen pariin. Toteutettujen ominaisuuksien ei tarvitse olla läpitestattuja, mutta kuitenkin konseptiltaan toimivia. Nykyohjelman parissa työskentelevät täydentävät ominaisuuksia. Vastaavasti myös nykyohjelman parista voidaan hetkellisesti siirtyä uudistamisen pariin toteuttamaan rinnakkaisia työvaiheita. Esitutkimuksen lopputuloksen on pystyttävä kertomaan, miten ja kuinka paljon työtä menee mihinkin osaan uudistamisprojektin aikana, jotta voidaan jakaa muut työtehtävät uudelleen.

Esitutkimuksen selkein ja näkyvin osa on muutoksen tyypin määrittely. Silloin kun halutaan pitää yhteensopivuutta yllä, toteutetaan sisäisen rakenteen uudistaminen (*redesign*). Sen perustana toimii aiempi vaatimusmäärittely ja se voidaan toteuttaa refaktoroimalla (*refactoring*). Sovelluksen suoritusympäristön muuttuessa myös ulkoisen rakenteen täytyy mukautua. Silloin mahdollisuuksina ovat ohjelman kääriminen (*wrapping*), uudelleenkirjoittaminen osittain (*rewriting*) tai kokonaan uudelleen tekeminen (*rewrite from scratch*). Kokonaan uudelleen tekemistä voidaan pitää poikkeuksellisena ratkaisuna, johon tulee ryhtyä vain erikoistapauksissa [8].

Yksi osa esitutkimusta on muutoskohteen määrittely. Yksinkertaisimmillaan kohde on koko ohjelma. Silloin joudutaan tilanteeseen, jossa kehitetään rinnakkain kahta ohjelmakokonaisuutta, joista toinen on kaikenlisäksi näkymätön asiakkaille ennen valmistumistaan. Yhtäkkinen uudistus (*big bang approach*) sisältää riskejä myös aikataulun kannalta. Ongelman minimoimiseksi on löydettävä kohtuullisen kokoisia osa-alueita uudistettavaksi (*Incremental approach*). Kerrosarkkitehtuurin mallia voidaan hyödyntää uudistettavan kohteen rajauksessa tiettyihin kerroksiin (hyödyllinen esim. tietokannan muuttuessa). Tilanteessa, jossa ei ole mahdollista suorittaa vanhaa ja uutta ohjelmaa rinnakkain (kokonaan uudenlainen sovellusalusta jne.), tulisi muutoksen aluksi käsittää vain tärkeimmät komponentit. Jokaisen läpiviedyn iteraation jälkeen uudistetaan edellistä vähemmän kriittisiä osia. Paloittain toteutettaessa pelkästään osion keskeisyys ei välttämättä ole kriteeri, vaan myös osioiden läheinen suhde toisiinsa voi olla syy toteuttaa muutokset tietyssä järjestyksessä. Esimerkiksi uudistettaessa ominaisuuden syöttötapaa, voi olla edukasta samalla tehdä raportointi, koska silloin voidaan toteuttaa uudistus siltä osin kokonaan. [1, pp. 171-174]

Löydettyä muutoskohdetta arvioidaan esitutkimuksessa muutamalla eri kriteerillä. Sille arvioidaan prioriteetti kuvastamaan tarpeellisuutta. Alustan muuttuessa tai teknistä etua tavoitellessa tarve on ehdoton ja sen myötä prioriteetti on suuri, mutta syyn

ollessa ohjelman rakenteessa tai ohjelmointikielessä on vaikeampaa perustella muutoksen tarpeellisuutta. Uudistamisesta saavutettavaa etua on usein vaikeaa arvioida tarkasti, joten siitä on tehtävä lyhyt selvitys. Muutoksen vaikutuksen laajuutta käyttäjäkuntaan täytyy myös puntaroida. Muutos, jolla on pieni prioriteetti ja siitä saavutettava etu vähäinen, ei välttämättä ole toteuttamisen arvoinen pienelle käyttäjämäärälle. Toisaalta taas pieni prioriteettinen muutos, joka hyödyttää suurta joukkoa, on toteuttamisen arvoinen [1, pp. 167-168].

Tärkeä osa päätöksenteon perusteita on myös riskien tunnistaminen [1, p. 174]. Aiemmin mainittujen resurssiongelmien lisäksi on tiedostettava ohjelmistoon pitkän elinkaaren aikana muodostunut sisäinen monimutkaisuus, joka on seurausta esimerkiksi virheiden korjauksista tai jälkeenpäin tehdyistä lisäyksistä. Ohjelmaan on saattanut tahattomasti muodostua komponenttien välille keskinäisiä riippuvuuksia, joita ei ole kuvattu missään, eikä niitä ole välttämättä tiedostettu. Riskien arvioinnissa on myös huomioitava kehittäjien kokemus uudesta sovellusympäristöstä, sillä tuntematon ympäristö voi tuoda mukanaan täysin uudenlaisia ongelmia, joihin ei ole valmiita vastauksia.

Esitutkimuksen lopputuloksena muodostuu selkeästi rajattu toteutussuunnitelma uudistamisen läpiviennistä toteutusvaiheineen ja aikatauluineen. Siihen liitetään ei-toiminnallisten vaatimusten muutokset ja mahdollisesti niistä aiheutuvat toimenpiteet (esim. käyttäjämäärän arvioiden perusteella palvelinhankinnat). Esitutkimuksen avulla tehdään päätös joko uudistamisen läpiviennistä tai sen hylkäämisestä. Kummassakin tapauksessa perustelut päätöksestä kirjataan, jotta jälkikäteen voidaan tarkastella näkemyksen oikeellisuutta tai käyttää apuna tulevissa uudistamisprojekteissa.

Aiemman määrittelyn ja esitutkimuksen perusteella luodaan uudistamisprojektin suunnitelma. Hyödyntämällä esitutkimuksessa havaittuja muutostarpeita määrittelyihin saadaan uudistetun ohjelmiston määritykset (kuva 3.9.). Samalla voidaan lisätä ja tarkentaa ohjelmiston ominaisuuksia, kuten modulaarisuus, testattavuus, tietoriippumattomuus ja lokalisointi [1, p. 170]. Ominaisuuksia valittaessa täytyy myös päättää, mitkä ovat niiden yleiset periaatteet. Esimerkiksi modulaarisuuden osalta voi olla tärkeää päättää ennen suunnittelua, miten hyvin ohjelma on jaettavissa osiin. Hyvin valitut ja selkeästi avatut ominaisuudet auttavat yhdenmukaisen toteutuksen tekemisessä.

4 TYÖKALUT KEHITYSYMPÄRISTÖSSÄ

Työkalut vaikuttavat hyvin paljon lopputulokseen, niin myös ohjelmistotuotannossa. Jokaisella työkalulla on omanlaisensa hyödyntämiskohde ja tapa tuottaa lisäarvoa kehitystyölle. Hyvien työkalujen lisäksi laatuun vaikuttaa erityisen paljon tapa, jolla niitä käytetään. Toimintatapoja tutkimalla voidaan löytää asioita, joiden avulla toimintaa tehostetaan ja virheiden määrä pienenee. Pitkän elinkaaren tuotteen kehityksessä tulee vastaan tilanteita, jolloin työkaluja täytyy arvioida kriittisesti. Syynä voivat olla alustaan liittyvät muutospaineet kuten ylläpidon päättyminen tai tuotteen koon kasvaessa suuremman automaatioasteen saavuttaminen.

Osiossa perehdytään kehityksessä käytettyihin työkaluihin käsitellen jokaista toiminnan osaa erillisenä kohteena. Toiminnan osat on jaoteltu lähdekoodin muokkaamiseen, tietokantojen kehitykseen, raportointiin, virheiden seurantaan ja versiohallintaan. Versiohallintaa ja versiointia tutkitaan omassa osiossaan sen laajuuden ja tärkeyden takia. Tarkoituksena on luoda käsitys kunkin osan erityisistä tarpeista, niiden edellyttämistä työkaluista ja niihin sovellettavista toimintamenetelmistä.

Käytetyin työkalu kehitystyössä yrityksessä on *Embarcadero Delphi XE* professional. Sitä käytetään laajimman ohjelmatason lähdekoodin luomiseen. Kolmen pienemmän ohjelman lähdekoodin muokkauksiin käytetään tuotteen aiempaa versiota, joka on nimeltään *Delphi 7*. Ne ovat *Object Pascalia* käyttäviä ohjelmointiympäristöjä työpöytäsovellusten tuottamiseen. Delphillä käännetyt ohjelmat on tarkoitettu toimimaan Windows-ympäristössä. Ohjelmointiympäristöön on mahdollista lisätä ulkopuolisten tekemiä suunnittelun aikaisia paketteja, joita on ilmaisia (esim. JEDI) ja maksullisia (esim. TMS component pack).

Delphin pääasiallinen toiminta keskittyy koodieditorina, debuggerina ja kääntäjänä toimimiseen. Ohjelmointiympäristön perustoimintojen lisäksi se sisältää profiloijan. Sen tarkoitus on analysoida ohjelmaa suorituksen aikana. Analyysin tuloksena saadaan tietoja suorittimen ajankäytöstä, muistivarauksista sekä kutsukaaviot.

Tietokantana käytetään alimmalla ohjelmatasolla Microsoftin Accessia ja muilla MS SQL Serveriä. Tietokannan taulut luodaan ja muokataan käyttäen niitä varten luotuja kuvaustiedostoja (Object Map File, OMF) käyttäen. Kyseisissä tiedostoissa on määritetty taulun sarakkeet ja niiden tyypit, indeksit sekä käytettävä skeema. Muuhun hallinnointiin ja sisällön tutkimiseen käytetään kyselytyökalua.

Tietokannan hallinnoimisessa voi apuna käyttää *Microsoft SQL Server Management Studio*ta. Se tarjoaa hieman tavallista kysely-syöte yhdistelmää graafisemman ilmeen. Lisäksi tietokannan triggereitä ja proseduureja voi tutkia suorituksen aikaisesti.

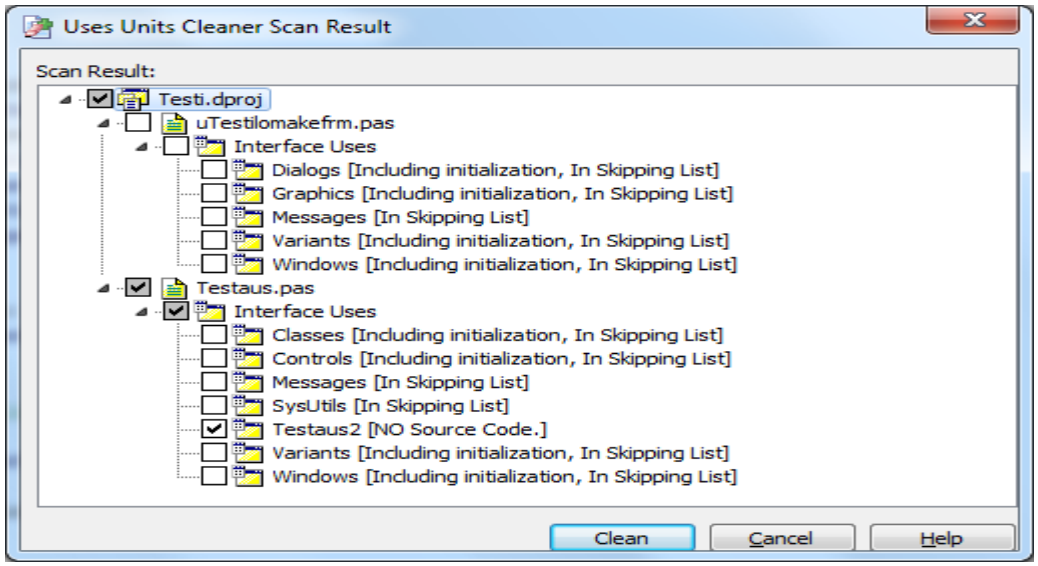
Raportoinnissa käytetään *FastReportsin* VCL-versiota. Se on raporttigeneraattori, jossa on visuaalinen suunnittelutila, jossa lisätään joukkioissa (band) tietoa raportin sivulle. Joukkio voi liittyä tiettyyn tietojoukkioon tai olla sivun vakio-osana. Raportin kenttien tietoa voi muokata PascalScriptin (voi käyttää myös C++Script:ä, JScript:ä ja BasicScript:ä) avulla suorituksen aikana.

Versiohallintana toimii tällä hetkellä *Team Coherence* niminen sovellus. Se sisältää tiedostojen ulos- ja sisäänkirjauksen, versioiden lisäämisen revisiolle sekä versiokohtaiset näkymät, jossa on mukana vain kyseisen version tiedostot. Versiohallinta on parhaillaan vaihtumassa. Syynä vaihdokseen on automatisoinnin tarve.

Uusia ominaisuuksia ja ohjelmasta havaittuja virheitä varten käytössä on *Flyspray*, virheidenseurantajärjestelmä (bug tracker). Se on ilmainen web-pohjainen PHP:llä toteutettu tehtävien kirjaus- ja seurantajärjestelmä.

4.1 Lähdekoodin analysointi ja hallinta

Jokainen ohjelmistotuote muodostuu joukosta lähdekooditiedostoja, jotka sisältävät luokat ja toiminnallisuudet. Tuotteen elinkaaren aikana tiedostoja tulee huomattava määrä lisää. Usein myös jossain vaiheessa tulee tilanteita, jolloin tiedostoissa oleviin luokkiin ja metodeihin ei enää viitata tai niitä suositella käytettäväksi. Sillä hetkellä, kun tiedostoon ei enää suoraan viitata, olisi se hyvä poistaa viittauksista. Käyttämättömien yksiköiden etsintä manuaalisesti tiedostoja läpi käymällä on melko hidasta. Hyvä apuväline käyttämättömien yksiköiden etsintään löytyy *CnPack IDE Wizard*:sta (kuva 4.1.), joka on ilmaiseksi ladattava lisäosa Delphin IDE:en. Apuohjelma osaa ehdottaa poistamaan yksiköitä, jotka eivät ole enää käytössä. Se osaa myös tutkia tiedostojen aloitukset (*initialization*) ja lopetukset (*finalization*). Alla olevassa tilanteessa (kuva 4.1.) on projekti, joka koostuu kolmesta yksiköstä *uTestilomakefrm*, *Testaus* sekä *Testaus2*. Sovellus esittää yksiköt, joihin ei ole suoraa viitettä, mutta merkitsee vain ne, joihin ei viitata missään osassa ja joita ei asetuksissa määritellä sallittavien listalle. Alla olevan kuvan 4.1. tilanteessa uTestilomakefrm- tai Testaus-yksiköstä ei viitattu Testaus2-yksikön luokkiin tai metodeihin.



Kuva 4.1. *Käyttämättömien yksiköiden etsintä.*

Viittausten poistaminen on siinä mielessä hyödyllistä, että käännetyn tiedoston koko pienenee. Koko pienenee kuitenkin vain, jos kaikki viittaukset kyseiseen yksikköön poistetaan.

Vanhojen viittausten poistaminen on hyödyllistä, mutta lähdekooditiedostot säilyvät edelleen. Itse tiedostot eivät haittaa ohjelmaa, mutta saattavat aiheuttaa sekaannuksia. Tilannetta voidaan lähestyä kahdella eri tavalla; joko poistamalla vanhentuneet tiedostot saman tien tai asettamalla niihin viesti esikääntäjältä.

Tiedostojen poistaminen on hyvin suoraviivainen ratkaisu ongelmaan. Sen selvänä etuna on, että kukaan ei käytä vanhentuneita luokkia ja toimintoja. Pienin poistamisesta aiheutuva ongelma voi olla se, että ohjelma ei enää käänny. Vika on helppo paikantaa ja usein myös käytetty toiminnallisuus korvata uudemmalla. Vaikeammin havaittava ongelma muodostuu silloin, kun poistettu yksikkö sisältää samannimisen metodin tai tyypin kuin joku jäljelle jääneistä yksiköistä. Kyseisessä tilanteessa ohjelmaan voi muodostua paha looginen virhe, joka voi jäädä huomaamatta. Esimerkkinä on tilanne, jossa yksiköt *Testaus* ja *Testaus2* sisältävät identtiset funktiot nimeltään *Pyoristys*, joka palauttaa annetun luvun pyöristettynä. Kummassakin funktiossa on samat parametrit. Ellei nimiavaruutta ole erikseen määritelty kutsuttaessa, viimeiseksi esitellyn yksikön funktiota käytetään. Esimerkkitilanteessa muodostuu ongelma siitä, jos kumpikin funktio käsittelee toiminnan omalla tavallaan, esimerkkitapauksessa lukujen pyöristäminen saattaisi aiheuttaa pieniä poikkeamia ohjelmaan.

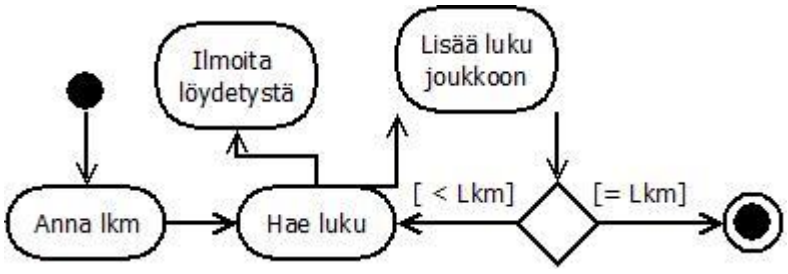
Kääntäjävaroitukset tai virheet ovat tiedostojen totaalista poistamista kevyempi ratkaisu. Käyttäessään vanhentunutta yksikköä kääntäjä antaa huomautuksen asiasta, johon kehittäjä reagoi. Kaikissa tilanteissa koko yksikkö ei ole kokonaan poissa käytöstä, vaan se sisältää osia, joita käyttäessä tulee huomioida asioita. Kyseisiin kohtiin viitatessa voidaan käyttää vihjeitä ilmoittamaan huomioitavasta tilasta. Jos kyseessä oleva toiminto on esimerkiksi riippuvainen alustasta tai kirjastosta, voidaan käyttää *platform*- ja *library*-määritteitä, vanhentuneelle tai kokeelliselle määreitä *depracated* tai *experi-*

mental. Mainitut vihjeet eivät suoraan estä virheitä, mutta havahduttavat kehittäjän ja mahdollisissa vikatilanteissa ohjaavat nopeammin ongelman lähteelle.

Pelkästään kääntäjälle toimiva lähdekoodi on vain osittain ylläpidetty. Toinen osa on sen selkeyden ja ymmärrettävyyden parantaminen kehittäjiä ajatellen. Yksinkertaisimmillaan se käsittää lähdekoodin muotoilun tiettyjen sääntöjen mukaan. Siihen tarkoitukseen Delphi:ssä on työkalu, joka hoitaa työn nopeasti. Myös sellaisia koodin osia on hyvä kommentoida, joiden toiminta ei ole selvää tai niihin liittyy joitain erityisvaatimuksia.

Suorasta lähdekoodin kommentoinnista tasoa voi nostaa luomalla tietohakemiston toiminnoista. Erityisen hyödyllinen tietohakemisto on tilanteissa, joissa on tarve käyttää jotakin yleiskäyttöistä toimintoa. Usein haluttu toiminto on olemassa, mutta joko ei tiedetä tarkalleen, missä se on tai mitä käyttötarkoitusta varten se on alun perin luotu. Toiminnon löytää etsimällä, mutta toteutuksen sisältö saattaa jäädä hieman epäselväksi. Tietohakemistoon voitaisiin lajitella toiminnot omiin kategorioihinsa ja kertoa niistä lisätietoa kuten raja-arvoja ja mahdollisia poikkeustapauksia. Hyvä toteutustapa tietohakemistolle on *Wiki*-tyyppinen palvelu, johon kehittäjät voivat lisätä toimintoja sekä muuta tietoa ohjelmasta.

Lähdekoodia voidaan analysoida suorituksen aikana, jolloin saadaan tarkempaa tietoa ohjelmakoodin käyttäytymisestä. Delphin mukana tulee *AQTime*-profiloija, jolla voi tutkia ohjelman suorituskykyä ja muistivarauksia metodeittain. Ohjelmalla on mahdollista tutkia lähdekoodia myös riveittäin, mutta se vaatii erillisen lisenssin. Profiloitavia kohteita on muistin- ja resurssienvarauksista suoritusajan mittauksiin. Profiloinnin raportti luodaan suorituksen aikana kerätyistä tiedoista ja näytetään automaattisesti ohjelman sulkeuduttua. Erityisen hyödyllinen profiloija on kattavuus-analyysi (*coverage analysis*). Sen avulla voidaan tarkastella, mitä ohjelman osia suoritetaan ja mitä ei. Raportti kertoo suoritusten määrän, katettujen rivien määrän ja osuuden joka suoritettiin. Ominaisuuden testaamista varten luodaan ohjelma, joka laskee käyttäjän antaman määrän alkulukuja. Sen aktiviteettikaavio on esitetty kuvassa 4.2.



Kuva 4.2. *Profiloinnin testiohjelman tilakaavio.*

Alla (taulukko 4.1.) on esimerkkituloste testiohjelman kattavuus-analyysistä. Tulos muodostuu ohjelmaa käytettäessä, joten myös tuloksia hyödynnettäessä on tiedettävä, mitä halutaan tutkia. Esimerkkiohjelmassa on kaksi toimintoa, joita ei käytetty ollenkaan, *btnAsiakasMuokkausClick* ja *btnLataaClick*. Se, että toimintoja ei käytetty tässä testissä, ei tarkoita ettei niihin kuitenkin voitaisi viitata.

Taulukko 4.1. *AQTime profiloijan kattavuus-analyysin tuloste.*

Routine Name	Hit Count	To-tal Lines	% Covered
TProfiloitavaLuokka::Create	1	0	100
LisaaTaulukkoon	10000	0	100
TProfiloitavaLuok-ka::LaskeAlkulukuja	1	0	100
TdlgProgress::FormCreate	1	0	100
TdlgProgress::IlmoitaLoydetysta	10001	0	100
TfrmTestiloma-ke::btnLaskeAlkuClick	1	0	100
TfrmTestilomake::FormCreate	1	0	100
TfrmTestilomake::FormDestroy	1	0	100
TfrmTestilomake::PaivitaMaara	10001	0	100
TfrmTestiloma-ke::btnAsiakasMuokkausClick	0	0	0
TfrmTestilomake::btnLataaClick	0	0	0

Ohjelman suorituskyvyn arviointi (*Performance analysis*) on myös hyvin oleellinen tutkimisen kohde, vaikkakaan ei kriittinen taloushallinnon sovelluksessa. Esimerkkiohjelman tapauksessa suuri osa ajasta menee lukujen etsimiseen. Lopputuloksessa esitetään jokaisen metodin suorittamiseen kulunut aika. Lisäksi ilmoitetaan suoritusten määrä. (taulukko 4.2.).

Taulukko 4.2. *AQTime profiloijan suorituskyky-analyysi.*

Routine Name	Time	Time with Children	Shared Time	Hit Count
TProfiloitavaLuok-ka::LaskeAlkulukuja	20,64528182	23,94721503	86,21161913	1
TdlgProgress::IlmoitaLoydetysta	3,299221199	3,299221199	100	10001
TfrmTestiloma-ke::btnLaskeAlkuClick	0,024323713	23,97202778	0,101467065	1
LisaaTaulukkoon	0,001758284	3,301933216	0,053250125	10000
TfrmTestilomake::PaivitaMaara	0,001442397	3,300663596	0,043700208	10001

Normaalitilanteessa taulukon 4.2. tulos on selviö. Analyysistä saa hyödyn jos siitä löytyy odottamaton poikkeama. Lisätään esimerkkiohjelman *LisaaTaulukkoon* -

metodiin ongelma (pysäytetään suoritus viideksi millisekunniksi) ja suoritetaan profilointi uudestaan (taulukko 4.3.).

Taulukko 4.3. *Suorituskyky-analyysi hidastetulla metodilla.*

Routine Name	Time	Time with Children	Shared Time	Hit Count
LisaaTaulukkoon	45,63853015	49,20200497	92,75746016	10000
TProfiloitavaLuokka::LaskeAlkulukuja	20,28963871	69,49164367	29,19723529	1
TdlgProgress::IlmoitaLoydetysta	3,555220172	3,555220172	100	10001
TfrmTestilomake::btnLaskeAlkuClick	0,047765704	69,53991756	0,06868818	1
TfrmTestilomake::PaivitaMaara	0,008762405	3,563982578	0,245859936	10001

Testin tuloksesta (taulukko 4.3.) havaitaan nyt selvästi, että samalla toistomäärällä toiminut metodi vei suorituksessa paljon enemmän aikaa. Todellisissa tilanteissa arviointi ei ole näin yksinkertaista, sillä arvioita on vaikea muodostaa. Profiloija sopii hyvin apuvälineeksi ongelmanselvitykseen.

4.2 Tietokantojen kehitys

Ohjelma käyttää tietovarastona Microsoftin SQL-Serveriä. (Kahdella pienemmällä ohjelmatasolla on Microsoftin Access-tietokanta.) Ajan myötä ohjelma kehittyy ja tarpeet tietokantaa kohden samalla kehittyvät. Koska kyseessä on asiakassovellus, ohjelman on huolehdittava tietokannan päivityksistä ja huollosta lähes automaattisesti. Rakennetta ylläpidetään ulkoisissa tiedostoissa, jotka sisältävät tiedon jokaisesta taulusta ja sen rakenteesta. Jokaisesta taulun sarakkeesta on kerrottu sen tietotyyppi, koko, yksilöllisyys, oletusarvo ja se, onko arvo pakollinen. Jokaisen taulun indeksit määritellään samalla omina tietoinaan tiedostoon.

Ohjelmiston päivityksen yhteydessä suoritetaan tarvittaessa tietokantojen päivitys, joka tarkoittaa sitä, että jokaista olemassa olevaa taulua muokataan ja uudet taulut luodaan. Olemassa olevien taulujen kohdalla tiedostossa määriteltyä rakennetta verrataan tietokannassa olevaan tauluun.

Tietokantojen päivityksessä tietokannat lukitaan, jotta kantaan ei tehtäisi päivityksen yhteydessä muutoksia. Lisäksi kirjautuminen järjestelmään sisälle estetään ja ilmoitetaan kirjautujalle meneillään olevasta päivityksestä. Tietokannan päivityksen onnistuessa muutoksen transaktio lopetetaan ja tietokannan versio päivitetään, jotta toiselta työasemalta ei päivitettäisi sitä turhaan.

Tietokantojen huoltoajossa parannetaan hakujen nopeutta ja tietokannan eheyttä. Se saavutetaan poistamalla vanhoja tietoja. Lisäksi tietokannan hallintajärjestelmää voidaan tehostaa. Yksi tehostuskeino on indeksien luominen uudestaan. Normaalisti tietokannan hallintajärjestelmä hoitaa indeksien parantelua taustalla, mutta jos järjestelmässä on paljon toimintaa, voivat indeksit olla vanhentuneita. Tällöin voidaan käyttää hakujen tehostamiseen sisäänrakennettua proseduuria luomaan indeksit uudelleen. Huoltotoimenpiteiden, joissa tietokannasta on poistettu tai lisätty paljon tietoa, kannattaa suorittaa kyselytilastojen uudelleenluonti [9]. Kyselytilastot ovat luotuja tilastoja siitä, miten tietoa voidaan hakea mahdollisimman tehokkaasti. Huoltoajon jälkeen myös tallennetut proseduurit kannattaa kääntää uudelleen, jotta ne olisivat mahdollisimman tehokkaita. proseduurit käännetään uudelleen joko seuraavan tietokannan käynnistyskerran jälkeen tai kun niillä ei ole suoritussuunnitelmaa [10].

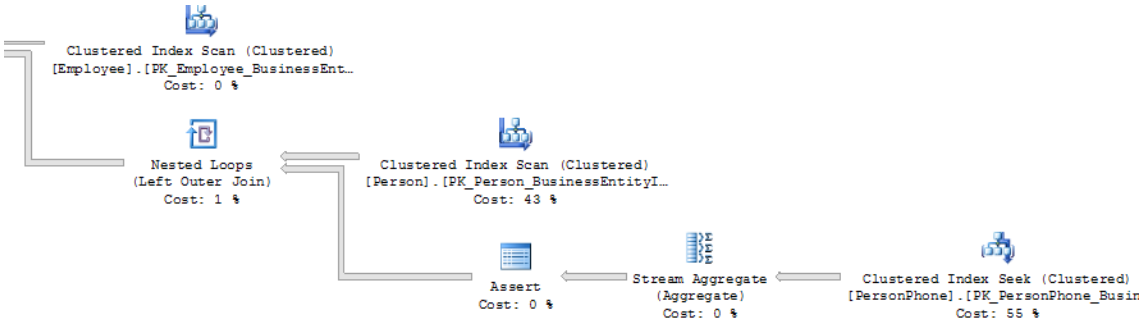
Tietokantakyselyitä luodessa kyselyn tehokkuutta on vaikea arvioida. Usein käytössä on vain suhteellisen pieni tietomäärä, jolla toimintaa testataan. Arviointiin voi käyttää apuna *Microsoft SQL Server Management Studio*:ta. Sillä suoritetuista kyselyistä voidaan tarkastella toteutussuunnitelmaa ja asiakaspuolen toteutuneita tilastoja. Toteutussuunnitelma sisältää vaiheittain jokaisen toimenpiteen sekä niiden kustannusarviot. Toteutuneiden tilastojen avulla voidaan tarkastella tutkia tapahtumia jaoteltuna kysely-, verkko- ja aikaosaan. Jokainen kysely suoritetaan kolme kertaa ja niistä esitetään myös keskiarvot. Suoritussuunnitelmaa ja asiakaspuolen tilastoja voi käyttää apuna testattaessa muutoksia olemassa oleviin ja uusiin kyselyihin, jotta saadaan selville muutosten vaikutukset tehokkuuteen ja toteutustapaan.

Microsoftin SQL-Serveriin löytyy testitietokanta *AdventureWorks*, jolla voi testata erityyppisiä kyselyitä. Luodaan aluksi tehoton testikysely, jolla haetaan useasta taulusta tietoa (ohjelma 4.1.)

```
Select P.FirstName, P.LastName,
(Select PP.PhoneNumber From Person.PersonPhone PP Where
PP.BusinessEntityID=P.BusinessEntityID) AS PhoneNumber,
(Select (Select PT.Name From Person.PhoneNumberType PT Where
PP.PhoneNumberTypeID = PT.PhoneNumberTypeID) AS PhoneNumber From
Person.PersonPhone PP Where
PP.BusinessEntityID=P.BusinessEntityID) AS Name, P.PersonType,
(Select HE.JobTitle From HumanResources.Employee HE Where
P.BusinessEntityID=HE.BusinessEntityID) AS JobTitle
From Person.Person P
```

Ohjelma 4.1. *Kysely sisäkkäisillä kyselyillä.*

Kyselyn tehokkuutta heikentää se, että siinä haetaan tietoja alikyselyillä. Ohjelma 4.1. esittää tavan, jossa yksittäisiä sarakkeen arvoja haetaan omalla erillisellä kyselyllä. Alla on esitetty osa suoritussuunnitelmasta (kuva 4.3.).



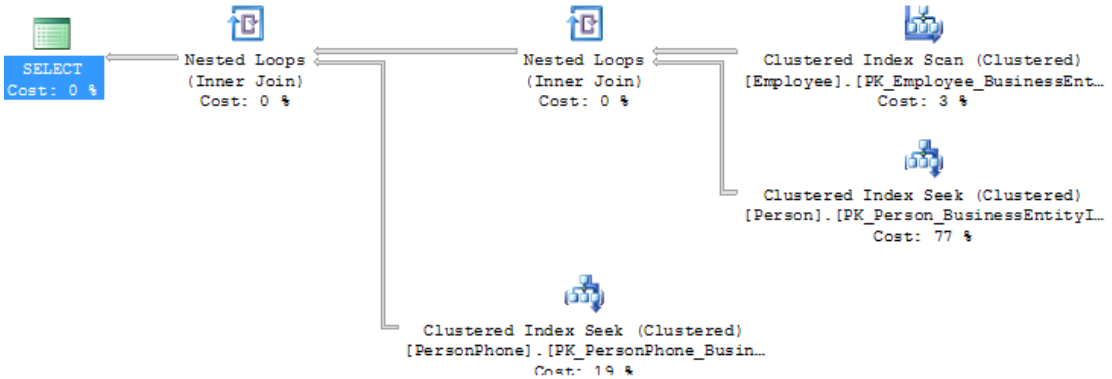
Kuva 4.3. *Sisäkkäisten kyselyiden suorituskaavio.*

Kyselyn tilastoa tutkimalla voidaan tarkastella yksityiskohtaisempia tietoja itse kyselyn suorituksesta. Koska nyt ensisijaisena tutkimuskohteena on kyselyn tehokkuus, tarkastellaan suoritusaikaa. Kyselyn (ohjelma 4.1.) kokonaissuoritusaika oli 1026ms. Kokonaisaika muodostuu palvelimen vasteajasta (aika viimeisen paketin lähettämisestä ensimmäisen saapumiseen) 92ms ja asiakkaan suoritusajasta (aika ensimmäisestä vastauksesta viimeiseen) 934 ms. Kyselyä muutetaan siten, että sisäkkäisen kyselyn sijasta tehdään tauluille liitos (ohjelma 4.2.).

```
Select P.FirstName, P.LastName, PP.PhoneNumber, P.PersonType,
HE.JobTitle
 From Person.Person P
 Inner Join Person.PersonPhone PP ON
(P.BusinessEntityID=PP.BusinessEntityID)
 Inner Join HumanResources.Employee HE ON
(P.BusinessEntityID=HE.BusinessEntityID)
```

Ohjelma 4.2. *Kysely käyttäen liitoksia.*

Muutoksen jälkeen Kokonaisaika oli 159ms, josta palvelimen vasteaika oli 113ms ja asiakkaan suoritusaika oli 46ms. Ero ajoissa johtuu tietokannan rakenteesta ja indekseistä. Tarkoitus ei ole tässä kohdassa tutkia syvällisesti syitä kyselyeroihin, koska kyseessä on testaukseen luotu tietokanta ja ensimmäinen kysely oli tarkoitushakuisesti luotu heikosti suoriutuvaksi. Suorituskaaviosta (kuva 4.4.) voidaan kuitenkin helposti todeta parannetun version sisältävän vähemmän välivaiheita kuin ohjelma 4.1., josta oli esitetty vain osa.



Kuva 4.4 *Parannetun kyselyn suorituskaavio.*

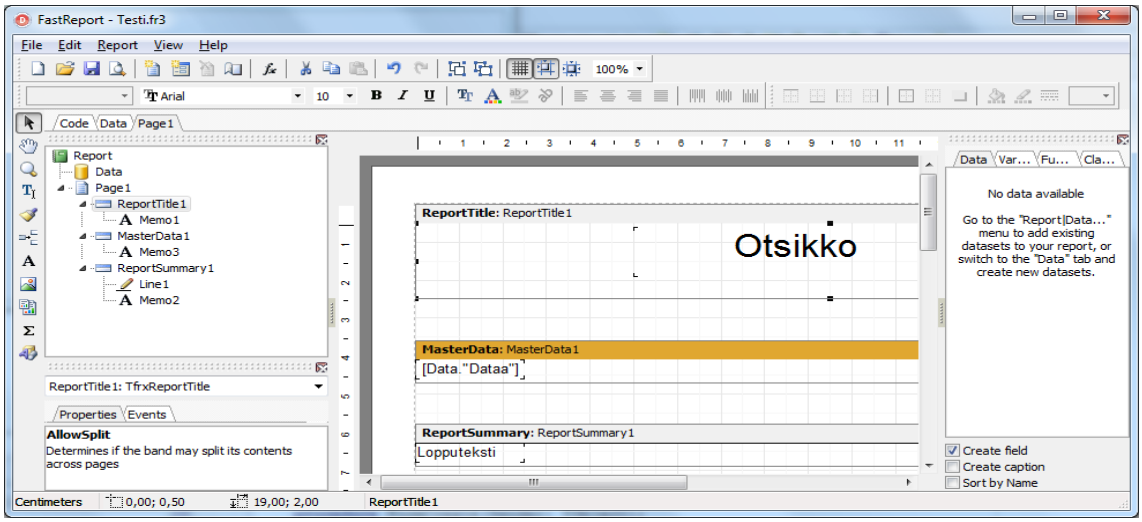
Kuvissa 4.3. ja 4.4. näkyvät solmukohdat sisältävät tarkemmin tietoa siinä suoritetusta toiminnasta. Tiedoissa kerrotaan fyysinen ja looginen operaatio, jotka kyseisessä kohdassa suoritettiin, sekä siitä aiheutuneen kustannuksen kokonaissuorituksesta. Solmua tutkimalla saadaan tarkempaa tietoa sen vaikutuksista I/O:n ja prosessorin käyttöön.

4.3 Raportoinnin työkalut

Raporttien tarkoituksena on järjestää ja koota tietoja halutulla tavalla selvästi luettavaan ja ymmärrettävään muotoon. Raporteista on saatava tietoa myös muihin sovelluksiin kuten taulukkolaskentaan. Raportoinnin työkaluilla on tarkoitus mahdollistaa edellä mainitut ominaisuudet. Näillä edellytyksillä toimii nykyisin käytössä oleva raporttimoottori *FastReport (FR).*

Ohjelmasta saatavat raportit muodostavat huomattavan osan ohjelman toiminnasta. Loppukäyttäjälle niistä saatavat tiedot ovat usein ensiarvoisen tärkeitä ja siksi niiden toiminta täytyy olla taattua.

FR:ssä raportit voi suunnitella joko lähdekoodissa tekstinä tai tallentamaan raporttipohjan mallin, joka on tehty graafisella suunnittelutyökalulla (kuva 4.5.). Usein on mieluisampaa tehdä raportti siten, että osan lopputuloksesta näkee välittömästi.

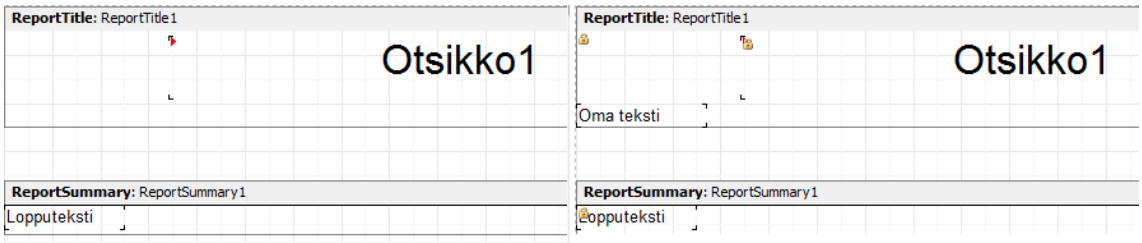


Kuva 4.5. *Raportin graafinen suunnittelutila.*

Suunnitteluohjelmalla luodut tiedostot voidaan tallentaa tiedostoiksi tai tietokantaan *BLOB*:na (Binary large object) [11]. Etuna tietokantaan tallennettaessa on se, että ohjelmaa ei tarvitse kääntää uudestaan raportin testaamista varten. Muutosten välitön näkyminen nopeuttaa raportin testausta ja parantelua, mutta mahdollistaa myös sen, että loppukäyttäjä voi muuttaa raportteja halutessaan. Muokattavuus ja hallinnan tunne ohjelmistosta edesauttavat hyvää käyttäjäkokemusta. Haittapuolena mahdollistuu kuitenkin sellainen tilanne, jossa raporteilta poistuu pakollisia tietoja tai pahimmillaan raportti lakkaa kokonaan toimimasta. Ongelma voidaan kiertää siten, että käyttäjän muokatessa raportteja ne eivät tallennu oletusraporttien päälle, vaan merkitsevät ne aina käyttäjän tekemiksi raporteiksi.

Raporttien hallinnointi yksittäisinä on aikaa vievää toimintaa. Muutokset, joissa muutetaan raporteille yhteisiä piirteitä, vaativat muutosta jokaisen raporttiin erikseen. Kehityspuolella raporttimallista on etua lähinnä kenttien sijoitteluun ja muotoiluun liittyvissä muutoksissa, sillä muuten tiedot haetaan suoraan tietokannasta. Rajoituksena perinnälle on se, että perittävän raportin on oltava normaalina tiedostona ja hakemistopolun on oltava tiedossa etukäteen.

Raportin periminen tapahtuu luomalla aluksi pohjaraportti, josta ominaisuuksia halutaan periä. Tämän jälkeen luodaan uusi raportti, jolle määritellään pohjaksi aiemmin luotu malli. Muokatessa perittyä raporttia mallista tulleissa ominaisuuksissa on lukon kuva, jolla ilmoitetaan myös siitä, että niitä ei voi muokata (kuva 4.6.).



Kuva 4.6. *Alkuperäinen (vasemmalla) ja periytetty raportti (oikealla).*

Periytyvien raporttien haittapuolena tallennusmuodon lisäksi on se, että perittyjä kohteita ei voi muokata vapaasti, vaan ne täytyy mennä muuttamaan alkuperäiseltä raportilta. Muokattavuuden väheneminen merkitsee myös aiemmin mainitun hallinnan tunteen pienenemistä. Ominaisuuden voi muuttaa toimimaan siten, että käyttäjän muokatessa raporttia poistetaan peritty raportti, jolloin käyttäjä voi tehdä haluamansa muutokset raporttiin. Tallennettaessa ohjelma luo tietokantaan käyttäjän muokkaaman raportin, joka ei periydy toisesta raportista.

Raporteille voi lisätä toiminnallisuutta liittämällä skriptejä raporttiin (ohjelma 4.5.). Fast-Reports:n skripti-kielen suurimpia rajoitteita on luokkien ja rakenteiden puuttuminen. Ulkoisiin luokkiin voi viitata, mutta niitä ei voi määritellä. Raportilta voidaan kutsua ohjelmassa olevia metodeita toteuttamaan toimintoja, jotka eivät raportilla välttämättä onnistu. Kutsuttavien metodien parametrien tulee olla perustyyppiä (*ordinaali*). Ulkoiset toiminnallisuudet luovat periytyvien raporttien kanssa mahdollisuuden tehdä raporteista yhtenäisen joukon, jossa toiminnallisuus on osittain eriytetty esitystavasta.

Esimerkkinä voidaan toteuttaa raporttiin ominaisuus, jossa tekstikenttää painamalla avautuu selaimessa haluttu www-sivu. Luodaan raportin OnUserFunction:lle *handleri* ja määritellään proseduuri *Linkki* suoritettavaksi sitä kutsuttaessa (ohjelma 4.3.).

```
function OnRaporttiUserFunction(const MethodName: string;
  var Params: Variant): Variant;
begin
if MethodName = 'LINKKI' then Linkki(Params[0]);
end;
```

Ohjelma 4.3. *Linkki-funktio.*

Linkki-proseduuri suorittaa verkkosivun avaamisen (Ohjelma 4.4):

```
procedure TfrmTestilomake.Linkki(Osoite: string);
begin
ShellExecute(Handle,'open', PWideChar(Osoite),nil,nil,
SW_SHOWNORMAL);
end;
```

Ohjelma 4.4. *Kysely Linkin avaamisen varsinainen suoritus.*

Raportilla määritellään tekstikentän *OnPreviewClick*-handlerissa kutsuttava sivusto ja suoritetaan proseduuri (Ohjelma 4.5.):

```
procedure OnPreviewClick(Sender: TfrxView; Button: TMouseButton;
Shift: Integer; var Modified: Boolean);
begin
  Linkki('http://www.google.fi');
end;
```

Ohjelma 4.5. *Linkin kutsuminen raportilta.*

Edellä (ohjelmat 4.3., 4.4. ja 4.5.) toteutetun toiminnallisuuden on tarkoitus esitellä käytön suhteellista helppoutta ja suoraviivaisuutta. Todellisessa käytössä raportti

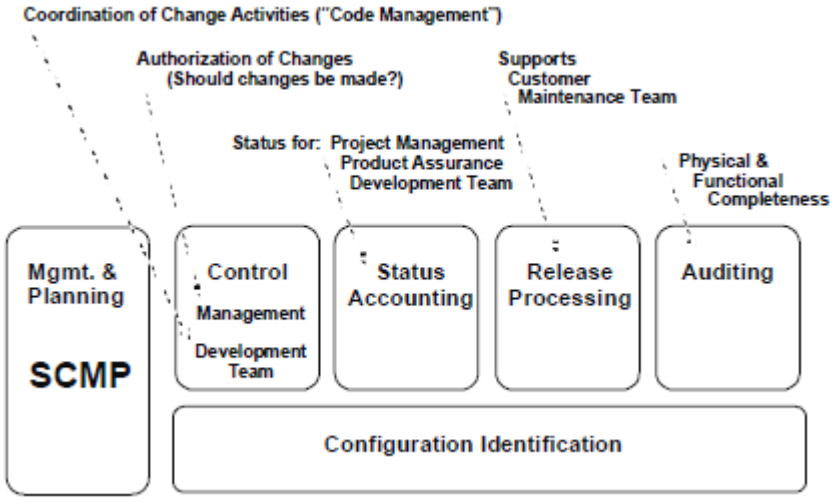
voisi pyytää lisätietoja tai avata kokonaan uuden raportin, josta näkyy paremmin halutut tiedot.

Raportilla oleville tekstikentille on mahdollista määritellä oma tyylinsä, joka sisältää kirjainkoon, värin ja muita tekstin olemukseen liittyviä määritteitä. Ohjelmaan linkitetyillä metodeilla tyylien muuttaminen on mahdollista, mutta se lisää työmäärää huomattavasti, koska silloin täytyisi käydä kaikki raportin komponentit läpi. Periyttämällä raportille on mahdollista määrittää tyylejä, mutta ne kentät joita ei periytettäisi, täytyisi muuttaa suoraan raportilla. Parempi ratkaisu on käyttää tyyli-kokoelmia [11], jotka sisältävät useita tyylejä. Ne voivat olla joko tiedostoissa tai tietokannassa. Raporttia luotaessa voidaan asettaa raportille viimeisin tyyli-joukko käyttöön, jolloin kaikki kentät joissa on määritelty jokin tyyli-joukon tyyleistä, muuttuisi määritellyn mukaiseksi. Käyttäjällä olisi valittavanaan joitakin malleja, mutta niitä pystyisi määrittelemään lisää. Tyyli-joukon pystyisi määrittämään raporttikohtaisesti, joten virallisemmat tulosteet voisi eritellä tyylimuutoksista.

5 TUOTTEEN KOOSTAMINEN

Suoritettavan toimivan ohjelman muodostaminen vaatii muutamia tyypillisiä työvaiheita. Jokainen ohjelmisto vaatii hallittua toimintamallia sitä tehtäessä, jotta lopputulos on halutunlainen. Hallittavuus tunnetaan ohjelmiston konfiguraation hallintana (*Software Configuration Management*). Hallinnan yhtenä apuvälineenä on versiohallinta, jota käsitellään omana osanaan. Valmistuotteen ollessa kyseessä integraatiotestausta ei voi suorittaa asiakaspäässä, joten se vaatii erityistä huomiota version muodostuksessa.

Konfiguraation käsitteeseen voidaan sisällyttää suunnitelmallisuutta eri osien yhteensopivuudesta [12, p. 51]. Ohjelmiston konfiguraatio ei ole pelkästään riippuvainen sen sisältämien lähdekooditiedostojen revisioista, vaan myös siihen liittyvien ulkoisten osien konfiguraatioista. Ulkoiset osat voivat olla laitteistoa tai niiden kiinteää ohjelmistoa (*firmware*). [13, p. 100] Kohdassa Konfiguraation hallinnan periaatteet (5.1) annetaan pohjustusta konfiguraation hallinnalle pohjautuen teoksen ”Guide to the Software Engineering Body of Knowledge 2004 Version” [13] konfiguraation hallinnan rakenteeseen (kuva 5.1.).



Kuva 5.1. *Konfiguraation hallinnan aktiviteetit [13].*

Tuotteen hallittavuudessa ja ylläpidettävyydessä siihen tehtyjen muutosten, sekä niiden välisten suhteiden ymmärtäminen on pohja kestävälle kehitystyölle. Versionhallinta ja ongelman seuranta ovat ohjelmistoalalla oleellisia työkaluja kehityksen lisäksi seurannassa ja arvioinnissa. Näiden toimintojen osalta tutkitaan käytössä olevia vaihtoehtoja ja niitä verrataan saatavilla oleviin.

5.1 Konfiguraation hallinnan periaatteet

Konfiguraation hallinta voidaan jakaa useisiin pienempiin osakokonaisuuksiin. Konfiguraation prosessin hallinta on ensimmäinen luonnollinen askel hallinnassa. Siinä määrittelyn kohteena ovat organisaation asettamat rajoitteet ja tavoitteet sekä määritellään seurannan parametrit, joita voivat olla erinäiset katselmoinnit ja mittaukset. Itse suunnitelmassa voidaan esittää käytettyjä työkaluja, resursseja ja aikatauluja sekä vastuualueita.

Konfiguraation tunnistamisen (*Configuration identification*) tarkoituksena on määritellä ne osat, joita ohjelman evoluution aikana halutaan seurata. Niitä voivat olla esimerkiksi lähdekooditiedostot, koodikirjastot tai raporttipohjat. Myös osien väliset suhteet ovat merkittäviä, sillä tuntemalla niiden riippuvuudet toisistaan voidaan ennakoida muutosten vaikutuksia läpi ohjelman. Konfiguraation tunnistaminen tarkoittaa myös tason (*baseline*) ja version tunnistamista konfiguraatioiden revisioiden joukosta. Jokaiselle revisiolle on mahdollista määritellä taso tai versio, johon se kuuluu.

Konfiguraation kontrolloinnilla (*Configuration control*) tarkoitetaan muutosten hallinnointia. Se käsittää periaatteet, joilla muutoksia voidaan tehdä sekä sen, miten niitä hyväksytään. Tekemällä mittauksia muutospyynnöistä ja hyväksytyistä muutoksista voidaan arvioida esimerkiksi ohjelmaan kohdistuneiden muutosten määrän trendiä, joka on kuitenkin hyödyllinen vain suuntaa antavana tietona, eikä tarkkana mittarina kypsyydestä. Myös muita muutoksia arvoinaan käyttävissä mittareissa on hyvä pitää niitä viitteellisinä lisätietoina, eikä varsinaisina kehitystyön mittapuina. [14, pp. 23-26]

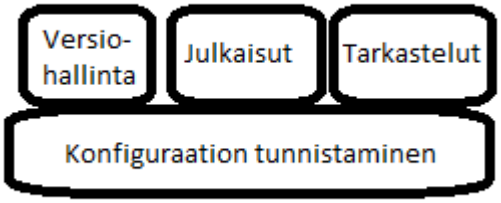
Ohjelman konfiguraation tilan tilittäminen (*status accounting*) on osa, joka jää helposti huomioimatta, vaikka se on aina tavalla tai toisella mukana. Se käsittää konfiguraation tilan tallentamisen ja sen tiedottamisen edelleen. Tilatietoa pidetään yllä automatisoiduin työkaluin. Tietoa tilasta voidaan välittää epävirallisesti tiimiltä toiselle, määräaikaisilla raporteilla tai kuten yleisesti ohjelmistotuotteissa on tapana, päivityksen yhteydessä omalla tiedotteellaan.

Konfiguraation tarkastus (*Software configuration auditing*) on muusta toiminnasta erillään oleva toimenpide. Tarkastus voi olla tyypiltään funktionaalinen (*FCA*), jolloin lähteenä on ohjelman verifiointi- ja validointi-aktiviteetit. Konfiguraation tarkastukseen kuuluu myös fyysinen tarkastus (*PCA*), jolloin tarkastellaan valmista tuotetta suunnittelu- ja määrittelydokumentteja vasten. Tarkastusten syynä voivat olla säädökset, standardit tai ohjeistukset. Esimerkiksi taloushallinnon ohjelmistoa on hyvä tarkastella muuttuneita verolakeja tai muita säädöksiä vasten. Tarkasteluväli kyseisen kaltaisessa ohjelmistossa voi olla vuosi, koska säädökset muuttuvat usein verovuosittain.

Ohjelman yhden konfiguraation iteraation viimeinen hallittava vaihe on julkaisu ja toimitus (*Software Release and Management Delivery*). Sillä käsitetään niin sisäiseen testaukseen kuin myös asiakkaille tarkoitetut julkaisut. Vaihe koostuu version luomisesta (*Building*) ja toimituksen hallinnasta (*Release Management*). Version luomisessa halutuista moduuleista (*software configuration items*) valitaan julkaisuun oikeat revisiot. Julkaisun hallinnassa ohjelmasta valitaan toimitettavaksi oikeat osat, sekä niihin liittyvä

dokumentaatio ja usein tiedot muutoksista. Jos kyseessä on esimerkiksi päivitysjulkaisu, ei kaikkia ohjelman tiedostoja tarvitse toimittaa asiakkaalle. Siinä tapauksessa myös päivityksen on tiedettävä esivaatimukset, jotta jokin aiempi korjaus ei jää tekemättä. Julkaisun hallinta voi sisältää myös toimituksen jälkiseurantaa, josta saadaan palautetta levityksessä tapahtuneista ongelmista.

Edellä mainitut kuusi konfiguraation hallintaan liittyvää vaihetta luovat pohjan sille, miten ohjelmistoa voi ja usein myös kannattaa kehittää. Henkilömäärältään pienen mittakaavan kehitysyhteisöissä malli on raskas käytettäväksi sellaisenaan. Eräs ominaisuus, jonka puolesta konfiguraation hallinnan mallia voidaan supistaa, on tuotteen yleiskäyttöisyys. Ohjelmistosta ei ole tehty asiakaskohtaisia versioita, joten variaatioiden määrä on vähäinen. Konfiguraation osat kavennetaan tunnistamiseen, versiohallintaan, julkaisuun ja tarkasteluihin (kuva 5.2.).



Kuva 5.2. *Yleistys konfiguraation hallinnasta.*

Yleistyksessä (kuva 5.2.) suunnittelu on osana konfiguraation tunnistamista. Raportointi on mallissa kevyempää kuin aiemmassa (kuva 5.1.) ja sitä toteutetaan pääosin versiohallinnan, konfiguraation tunnistamisen yhteydessä sekä ongelmanseurannassa.

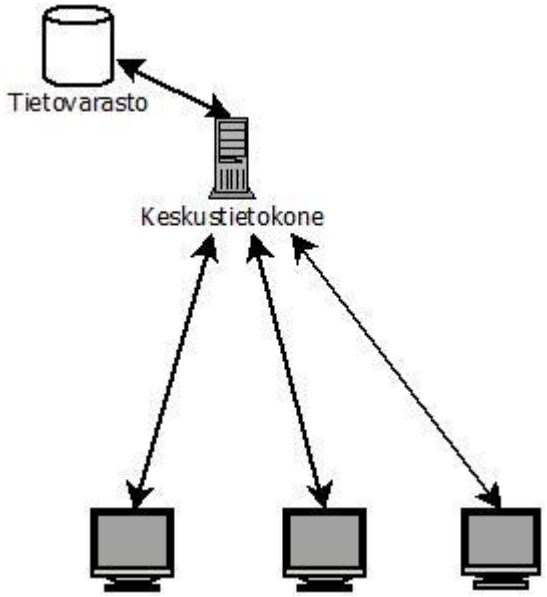
Kuvan 5.2. mukaisessa versiohallinnassa konfiguraatio-alkioiden revisioiden lisäksi ylläpidetään yhteensopivuudesta, ominaisuuksista ja kehityssuunnitelmista. Näin ollen versiohallinnan toiminta lähenee tuotetiedon hallintaa (*PDM*), antaen kuitenkin näkymän tulevaisuuden kehityksen suuntaan. Mallissa julkaisun osuudessa suunnitellaan ja valitaan revisioiden joukosta halutut. Ennen asiakasjulkaisua tehdään integraatiotestaus jaettavalla paketilla, jotta mahdolliset ongelmat havaitaan ja voidaan vielä korjata. Tarkastelut tapahtuvat määräajoin tai tarvittaessa. Niissä määritellään ohjelmaa koskevat mahdolliset muutostarpeet. Jokaisessa tarkastelussa tulisi tutkia, onko uusia kohteita tai onko jokin vanha kohde sellainen, että sitä ei tarvitse enää tarkastella. Tarkastelun lopputuloksena ohjelman etenemissuunnitelmaa muutetaan. Jokaisessa vaiheen tuotoksia tarkastellaan konfiguraation osina ja ne lisätään osaksi konfiguraatiota.

5.2 Versiohallintajärjestelmät

Versionhallintajärjestelmä on yksi osa toimivaa konfiguraatioiden hallintaa. Versiohallinta on konfiguraatio-alkioiden hallinnointia. Hallinta sisältää tiedostojen tilatiedon eli version (revision). Alkeellisimmillaan se tarkoittaa käyttäjän itse hallittavaa tiedostojen tallennuspaikkaa. Kyseinen menetelmä ei tarjoa automaattista tapaa seurata tiedostojen

evoluutiota, vaan tallennettuna on ainoastaan jokaisen tiedoston viimeisin versio. Erityisen ongelmallinen hallinnasta tulee silloin, kun kehittäjiä on enemmän kuin yksi. Silloin kehittäjien samanaikaisesti tekemät muutokset tiedostoihin voivat korvaantua ilman tietoa siitä, jos tiedostoa muokatessa joku muukin on tehnyt siihen muutoksia.

Ensimmäiset versiohallintasovellukset toimivat paikallisia tiedostoja muokkaamalla (Source Code Control System SCCS ja Revision Control System RCS). Nämä myös *Local only* mallina (kuva 5.3.) tunnetut versiohallinnat perustuivat keskuspalvelimiin, jossa tieto sijaitsi vain keskitetysti. Niiden keskeinen parannus oli vain osittainen tiedon tallennus eri revisioista. Näitä osittaisia muutoksia revisioiden välillä kutsutaan *deltoiksi*. Vain viimeisin revisio sisältää koko tiedoston. Aiempien revisioiden tiedostot muodostetaan deltojen avulla. SCCS ja RCS sisälsivät useita tiedostoista koostuvia kehityslinjoja. Näitä rinnakkaisia kehityslinjoja kutsutaan haaroiksi (*branch*). Tiedostojen useat haarat mahdollistivat esimerkiksi eri versiotasojen luonnin samasta tietovarastosta ja silti mahdollisuuden jatkaa jokaisessa ylläpitoa. [15]

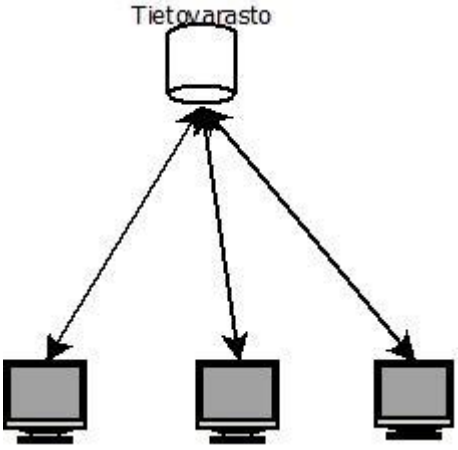


Kuva 5.3. *Local only-versiohallintamalli.*

Erillisten työasemien yleistyessä RCS:n päälle skripteillä rakennettu (myöhemmin uudistettu toimimaan ilman RCS:ää) CVS (*Concurrent versions system*) korvasi lähes kokonaan aiemmat järjestelmät. Se muutti versiohallinnan tavan käsitellä tiedostoja erillään olevissa sijainneissa samanaikaisesti. Lisäksi muutoksia käsiteltiin yksittäisten tiedostojen sijaan koko projektin osalta. Samalla myös revisioiden haaroittaminen voitiin kohdistaa koko projektiin yksittäisten tiedostojen sijaan. CVS:ää käytetään edelleen, mutta sen pitkän elinkaaren takia se ei täytä nykypäivän vaatimuksia samanaikaisesta työskentelystä ja tehokkuudesta. Se loi pohjan nykyisille versiohallinnoille ja monia siinä esiteltyjä ominaisuuksia sovelletaan uudemmissa versiohallintajärjestelmissä. CVS:ää voidaan pitää ensimmäisenä nykyisen kaltaisen keskitetyn versiohallintamallin toteuttajana. [15]

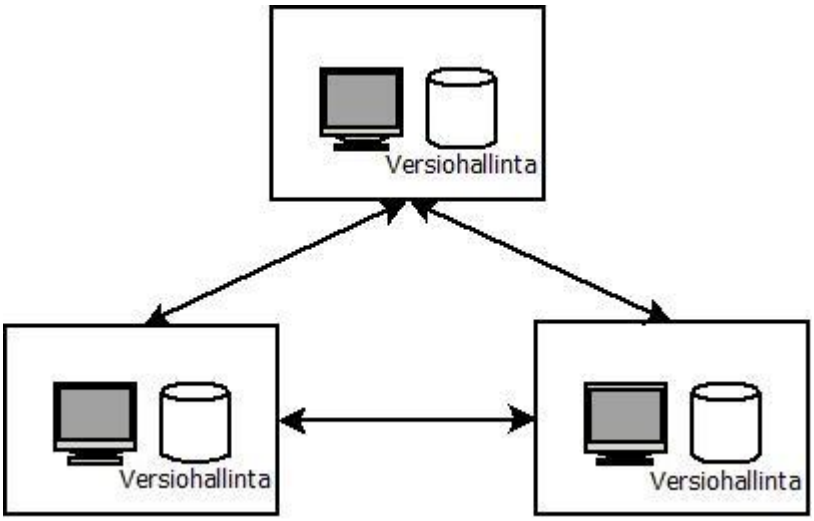
Uudemmissa versiohallintajärjestelmissä tärkeimpänä uudistuksena ovat atomiset operaatiot (*atomic operations*). Niillä tarkoitetaan sellaista operaatioiden joukkoa, joka suoritetaan kaikkien osalta onnistuneesti tai ei ollenkaan, tästä nimi atominen (jakamaton). Atomisia operaatioita käytetään uudemmissa järjestelmissä usein muutosten kirjoittamiseen jaettuun sijaintiin.

Keskitetyssä versiohallinnassa (kuva 5.4.) järjestelmän rakenne on Client-Server tyyppinen. Ne pitävät yllä tietovarastoa sen hyödyntäjän kannalta keskitetyssä sijainnissa. Kehittäjä ottaa työkopion keskitetystä sijainnista ja samalla järjestelmästä riippuen asettaa lukituksen. Joissain järjestelmissä lukitus osoittaa muille, että tiedoston parissa ollaan työskentelemässä, eikä se estä muita lukituksia. Muokkausten jälkeen lukitukset puretaan ja muutoksista luodaan uusi revisio.



Kuva 5.4. *Keskitetty versiohallinta.*

Hajautetut versiohallintajärjestelmät (*distributed version control system DVCS*) pyrkivät muuttamaan näkemystä Client-Server -mallista moniulotteisempaan rakenteeseen (kuva 5.5.). Kaikille hajautetuille versiohallintajärjestelmille on yhteistä tehokkaat yleiset operaatiot, koska ne suoritetaan paikallisesti.



Kuva 5.5. *Hajautettu versiohallinta.*

Eri haarojen yhdistäminen on myös hajautetuissa ratkaisuissa hyvin yleinen operaatio, koska kukin kehittäjä ylläpitää omaa versiohallintaansa. Eri tietovarastoja ei välttämättä yhdistellä yhtä usein kuin keskitetyissä ratkaisuissa.

5.3 Team Coherence - ja Kiln - versiohallintajärjestelmät

Nykyisin käytössä oleva versiohallintasovellus *Team Coherence* edustaa keskitettyä mallia. Siinä jokainen kehittäjä ottaa itselleen työkopiot haluamistaan tiedostoista. Ennen muutosten tekemistä tiedostot lukitaan ja otetaan muokkaukseen (*check-out*). Lukitustoiminto ei estä muita käyttäjiä lukitsemasta tiedostoa, mutta on merkki siitä, että samaa tiedostoa ollaan muokkaamassa toisaalla samanaikaisesti. Muutosten jälkeen tiedostot palautetaan tietovarastoon (*Check-in*). Palautettaessa tiedostoa muokkauksesta sille voidaan määritellä versio-nimike (*version label*, tunnetaan myös nimellä *tag*). Kyseinen nimike voi olla vain yhdellä versiolla. Yhdelle revisiolle voidaan määritellä useita versioita.

Team Coherencen etuna on sen kyky ymmärtää Delphin lähdekooditiedostoja hyvin. Lomaketiedostot ja datamoduullit esitetään yksittäisinä tiedostojen hallintanäkymässä. Ohjelma on mahdollista sulauttaa Delphin IDE:n, mutta käytössä se on aiheuttanut epävakautta itse IDE:n toimintaan ja aiheuttanut satunnaisia ohjelman kaatumisia. Sulautus suoraan kehitysympäristöön tuo joustavuutta käytettävyyteen, mutta hyöty epävakauteen verrattuna on pienempi. Team Coherencessa muutoksia käsitellään tiedostokohtaisesti, eikä muutosjoukkona projektikohtaisesti. Muutosjoukkojen käsittely yksittäisinä tiedostoina mahdollistaa tilanteita, joissa tehty muutos ei tulekaan seuraavan version mukana kokonaisuudessaan, koska joku muokatuista tiedostoista ei sisällä oikeata versio-merkintää muutetussa revisiossa.

Uudeksi versiohallinnaksi Team Coherencen tilalle on suunniteltu hajautetun mallin *Kiln*ia, joka pohjautuu *mercurial*iin. Mercurialin lähtökohtana ovat käytön helppous, skaalautuvuus ja hyvä suorituskyky. Muutoksia käsitellään koko projektin osalta, eikä vain yksittäisten tiedostojen kannalta. Revisioiden delttoihin tallentuu vain ero aiempaan revisioon. Binääri-tiedoston muuttuessa se täytyy tallentaa kokonaan uudestaan. Eri revisioiden välillä toimimista on pyritty nopeuttamaan lisäämällä delttojen joukkoon kokonaisia tiedostoja. Menetelmällä pyritään nopeuttamaan toimintaa, jos revisioita on paljon.

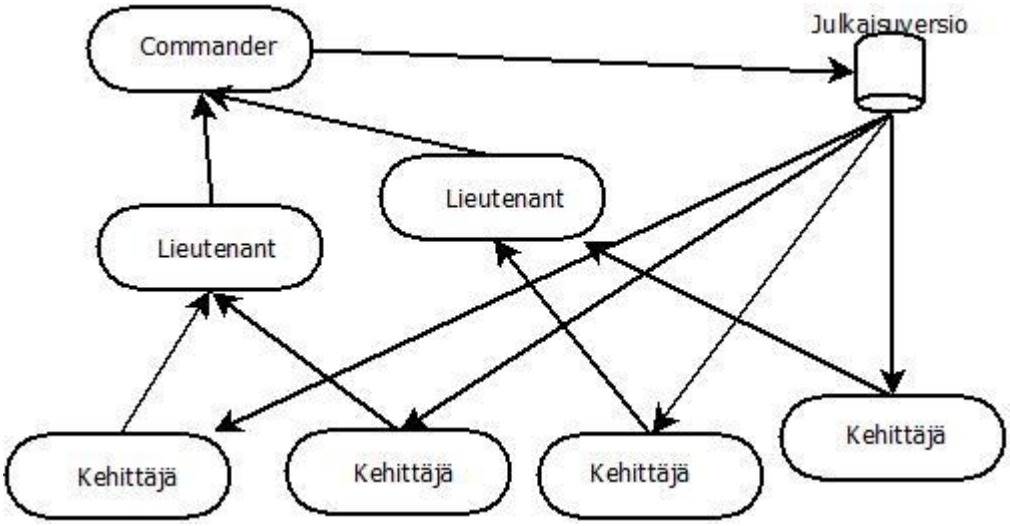
Monet Kilnin parannukset liittyvät käyttöliittymään ja käytettävyyteen, kuten paranneltu *DAG* (Directed Acyclic Graph) ja tietovarastojen ryhmittely. Eräs merkittävä parannus on kuitenkin laajennus binääritiedostojen käsittelyyn. Normaalisti binääritiedostoista tallennetaan jokainen muutos. Poistetut tiedostotkin pysyvät mukana, koska koko historia seuraa jokaista tietovarastoa. Kilnin laajennos mahdollistaa vain osittaisen binääritiedostojen historian mukana kuljettamisen. Toimintaa nopeutetaan paikallisella välimuistiin tallennetuilla historiatiedoilla. [16]. Toinen hyödyllinen lisätty ominaisuus

on mahdollisuus lisätä versio-tag suoraan Kilnin käyttöliittymän kautta tekemättä siitä ensin paikallista kopiota.

5.4 Hajautetun versiohallinnan työmallit

Hajautetussa mallissa tietovarastoilla voidaan rakennella hierarkioita. Keskitetyissä versiohallintajärjestelmissä hierarkioiden rakentelu ei ole tehokasta. Erityyppisiin hierarkioihin liittyy aina omat työmallinsa. Usean kehittäjän ympäristöissä yleisesti käytettyjä malleja ovat *Commander/Lieutenant* (kuva 5.6.), *Integration manager* (kuva 5.7.) ja perinteinen keskitetty malli (kuva 5.4.). Jokaisella mallilla on omat vahvuutensa tietynlaisissa tilanteissa ja työympäristöissä.

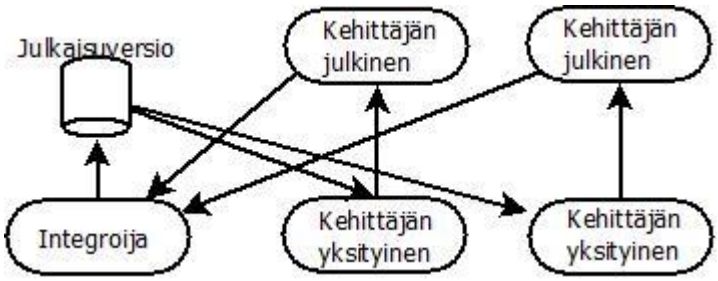
Keskitetyssä mallissa käytetään samoja menetelmiä kuin toimiessa varsinaisen keskitetyn järjestelmänhallinnan kanssa. Jokainen kehittäjä tekee muutokset yhteen keskitettyyn sijaintiin ja ottaa muiden muutokset sieltä. Toimintatapa sopii sellaisiin kehitysympäristöihin, jossa tuotteesta on kehityksessä ja ylläpidossa vain yksi versio ja kaikki kehittäjät tekevät kaikkia ohjelman osia.



Kuva 5.6. *Commander/Lieutenant työmalli.*

Commander/Lieutenant – mallissa muutokset kasataan kootusti tietyistä lähteistä alapuolelta. Julkaisuversion koostaja ei ota suoraan alimman tason toimijoiden muutoksia. Komentaja koostaa muutoksista julkaisuversion (käytetään myös nimitystä *blessed repository*). Käytännön elämässä kyseinen malli on käytössä Linuxin kernelin kehityksessä. Siinä kehittäjien suhde määräytyy maineen perusteella, jota on tullut aiemmista muutoksista ja korjauksista. Maine on riippuvainen myös alueesta, jota muutokset koskevat.

Commander/Lieutenant -työskentelymalli sopii suuriin projekteihin, joissa vastuualueita on jaettu eri kehittäjien kesken. Mallin käyttö vaatii sitä hyödyntävän organisaation olevan rakennettu selvästi hierarkisesti tai vastuualueiden hyvin tarkasti määriteltyjä. Pienelle organisaatiolle malli on turhan raskas muodollisuutensa takia.



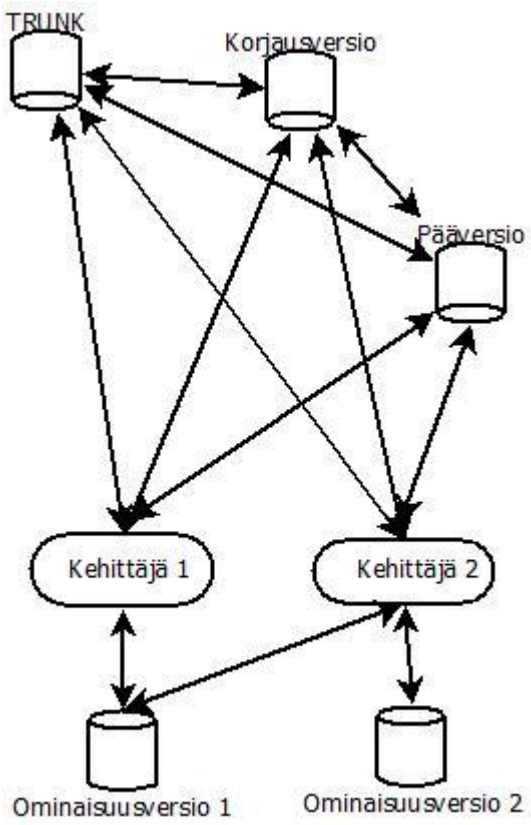
Kuva 5.7. *Integroija työmalli.*

Edellä esitellyt mallit (keskitetty, commander/lieutenant ja integroija) ovat yleisesti käytössä olevia. Ne eivät sovellu hyvin malliin, jossa kehittäjillä on useita vastuualueita ja samanaikaisia eritasoisia ohjelmaversioita kehityksessä. Lisäksi tiimin pieni koko vaikuttaa siihen, että malli ei saa olla liian jäykkä muuttumaan tarvittaessa.

Työympäristöön yhteensopivassa mallissa kaikki kehittäjät ovat yhdenveroisia tietovarastojen suhteen. Julkaisuversion lisäksi on ennakoitava tulevia vikakorjauksia ja uusia ominaisuuksia omilla tietovarastoillaan. Suuremmat muutokset voidaan toteuttaa omissa tietovarastoissaan, jotta testaamattomia ominaisuuksia ei tule vahingossa mukaan julkaisuversioissa.

Edellä kuvailtu malli esitetään havainnollistettuna mukautettuna työmallina kuvassa 5.8. Kuvan TRUNK-edustaa julkaisuversiota, johon tehdään suuren prioriteetin korjaukset. Toimintatapa mahdollistaa sen, että kesken normaalin päivitysaikataulun ohi voidaan tehdä uusi versio. Pääversio ja korjausversiot ovat tulevia julkaisuja varten luodut tietovarastot. Normaalisti versiota tehtäessä yhdistetään joko korjausversion tai pääversion muutokset TRUNK:n kanssa. Samalla annetaan oikea versio-tag viimeisimmälle mukaan tulevalle muutokselle. Siirryttäessä uuteen pääversioon voidaan vanha pääversio arkistoida.

Mallissa esitetyt ominaisuusversiot viittaavat päähaarasta (trunk) johdettuihin omiin kehityspolkuihin (*feature branch*). Niitä voi olla useita ja niiden parissa voi työskennellä yksi tai useampi kehittäjä. Kehityspolut sisältävät suurempia ohjelmakokonaisuuksia, joiden toteuttamisen kestosta ei ole tarkkaa tietoa, joten niitä ei kannata kiinnittää suoraan mihinkään pääversioon. Ominaisuuden tyypistä riippuen voidaan kehitystyön aikana yhdistää ohjelman päähaarasta muutoksia siihen, jotta ominaisuuden valmistuttua yhdistettävien muutosten määrä pysyisi suhteellisen pienenä. Muutoksia ei kuitenkaan kannata yhdistää, jos uusi ominaisuus tekee paljon muutoksia nykyiseen toiminnallisuuteen. Silloin on yhdistettäessä vaarana, että tehdyt korjaukset eivät toimi niin kuin oli tarkoitettu. Ongelmia on mahdoton välttää, mutta hyvillä toimintatavoilla ja selkeillä suunnitelmilla voi pienentää haittavaikutuksia. Ominaisuuksien valmistuttua ja niiden yhdistämisen jälkeen ne ovat osa päähaaran historiaa, eikä erillistä tietovarastoa tarvitse tallentaa.



Kuva 5.8. *Mukautettu työmalli.*

Esitetyn mallin (kuva 5.8.) heikkoutena on se, että sillä ei pystytä tukemaan rinnakkaisia ohjelmatasoja. Eri tasojen tukeminen vaatisi ohjelman rakenteeseen eri ohjelmakerrosten ohittamisen estämisen (kuva 2.3.). Ohjelman nykyiseen rakenteeseen sovitettuna olisi vaikeaa todeta, mihin ominaisuuden voi ohjelmassa lisätä suoraan vaikuttamaan muihin tasoihin. Ainoana vaihtoehtona on monistaa malli erikseen jokaiselle ohjelmatasolle ja suorittaa testaus yhden ohjelmaversion osalta kerrallaan. Silloin muutosten hallittavuus pysyy kohtuullisena.

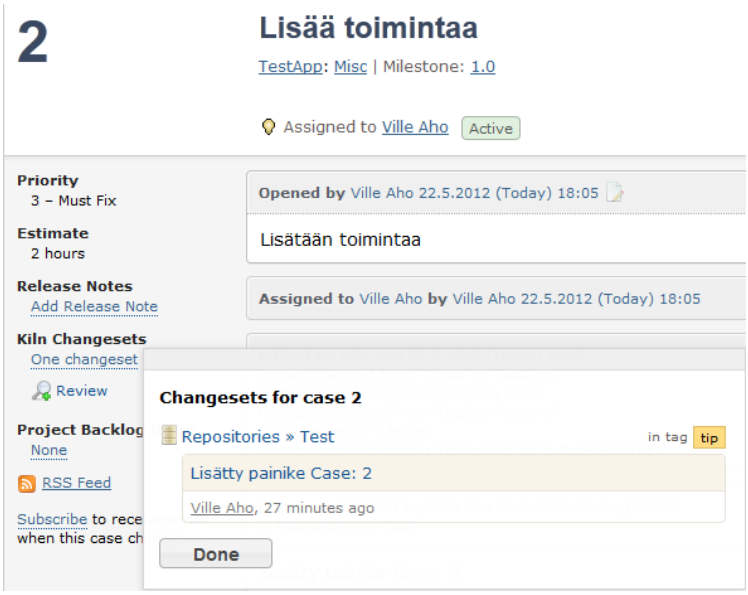
5.5 Muutosten seuranta

Konfiguraation kontrolloinnin yhtenä osana on muutosten seuranta. Kaikki ohjelmistoon kohdistuvat muutostarpeet täytyy tallentaa, jotta niiden perusteella voidaan myöhemmin arvioida muutosten vaikutuksia, tehdä versioiden muutosluetteloita ja löytää mahdollisten ongelmien alkulähteitä. Muutosten seurantaa voidaan harjoittaa yksinkertaisimmillaan taulukkoon kirjatuilla tiedoilla, mutta nykyisin on paljon tarkoitukseen soveltuvia ohjelmistoja. Kaikki näiden ohjelmistojen keskeisenä osana on tapahtumien kategoriointi, vastuuhenkilön tai -ryhmän määrittely ja tilan muuttaminen.

Tällä hetkellä muutostenseurannassa on käytössä *FlySpray*-ohjelmisto. Se on PHP:llä toteutettu ilmeinen ohjelmisto. Kirjatut muutostehtävät on mahdollista tallentaa MySQL tai PGSQL tietokantaan. Kategorioinnin, tilan ja vastuuhenkilön määrittelyn lisäksi tehtävät on mahdollista lajitella projektikohtaisesti. Tehtäviä voi myös halutessaan seurata olematta vastuuhenkilö, jolloin sen muuttuessa saa ilmoituksen. Tehtäville

on mahdollista määritellä myös versio, jolloin muutoksen odotetaan olevan valmis. Flyspray on hyvä ohjelmisto ollakseen ilmainen, mutta sen haittapuolena on siihen liittyvän tuen ja kehitystoiminnan puuttuminen. Viimeisimmästä version julkaisusta on kulunut yli kolme vuotta. Pitkä aikaväli julkaisuiden välillä ei suoranaisesti ole ongelma, mutta saattaa olla tietyissä tapauksissa tietoturvariski. [17]

Versiohallinnan muuttuessa Kiln:ksi hyväksi vaihtoehdoksi uudeksi muutosten seuranta-sovellukseksi muodostui *FogBugz*. Syynä FogBugz:n suosimiseen Kiln:n kanssa on niiden yhteistoiminta. Muutoksia tehtäessä voidaan määrittää yhteys FogBugz:n tehtävään. Muutokset näkyvät suoraan tehtävällä, jolloin niitä voi tarkastella kootusti yhdestä sijainnista (kuva 5.9.). Tarvittaessa voi pyytää katselmointia muutokseen, jolla voidaan ilmoittaa tiimille muutoksesta tai pyytää tarkastamaan muutoksen oikeellisuuden.



Kuva 5.9. *Tehtävään kohdistuvat muutokset.*

Fogbugz mahdollistaa muutosten seuraamisen monipuolisesti. Se ei silti automaattisesti osaa kategorisoida ongelmia tai asettaa niitä sopivan henkilön vastuulle. Yleinen pulma ongelmien ja ominaisuuksien lisäämisessä seurantajärjestelmään on niiden aikatauluttamisessa. Kirjattaessa muutospyyntöä on usein mahdotonta arvioida tunnin tarkkuudella sen toteutuksen kestoa. Mercurialin kanssa toimiessa työskentelyä helpottaa, jos etukäteen voidaan arvioida sitä, missä versiossa mikäkin ominaisuus on tai milloin ongelma korjataan. Virhearvioinneista saattaa koitua pitkiä viivästyksiä julkaisuun, koska olemassa olevien muutosten peruuttaminen vaatii tarkkuutta ja saattaa epäonnistuessa aiheuttaa uusia virheitä. Hyvä keino välttää virhearviointeja on jakaa muutospyyntö useampaan alijaksoon ja antaa niille arviot. Menetelmä ei estä viivästyksiä, mutta niihin voidaan varautua huomattavasti aiemmin ja mahdollisesti resursoida kehitystä sen mukaan. Toinen aikataulutukseen liittyvä ominaisuus on testauksen vaatiman ajan liittäminen muutoksen kokonaisaikaan. Paljon testausta vaativissa muutoksissa se kannattaa lisätä alitehtävänä varsinaiselle tehtävälle.

6 PÄÄTELMÄT

Jokainen ohjelmisto on sekoitus erilaisia arkkitehtuurityylejä. Jotkin tyylit ovat helpommin havaittavista rakenteesta kuin toiset. Tärkeintä on tunnistaa toteutuksesta tyylille ominaiset ratkaisut ja pyrkiä noudattamaan niihin liittyviä hyviä toimintaperiaatteita.

Kerrosarkkitehtuurissa jokaisella kerroksella on suotavaa olla omaa toiminnallisuutta. Ilman toimintaa oleva kerros aiheuttaa ylimääräistä työtä tekijöille. Yhtä tärkeää kuin on huolehtia kerrosten oleellisuudesta, on luoda kerrosten välille riippuvuuksia harkintaa käyttäen. Huonosti suunnitellut riippuvuudet aiheuttavat mahdollisesti ongelmatilanteita.

Tietovarastot ovat ohjelman eri osioita yhdistäviä rakenteita. Niiden tarkoituksena on varmistaa tiedon säilyvyys ohjelmasta riippumatta. Säilyvyyden lisäksi tiedon saatavuus on merkittävä ominaisuus tietovarastolle. Usean käyttäjän järjestelmissä tietovaraston on pystyttävä huolehtimaan lukituksista ja niiden vapautuksista. Suuremmissa muutoksissa tietovaraston tietoja voidaan muuttaa transaktio – pohjaisesti, jolloin muutokset voidaan vielä perua hallitusti. Eri ohjelmatasot ovat vahvasti riippuvaisia niihin liitetyistä tietovarastoista. Mahdollisuutta tietovaraston tyypistä riippumattomaan toimintaan täytyisi tutkia.

Sovelluksissa erinäiset komponentit välittävät tapahtumista tietoa niitä kuuntelemaan rekisteröityneille. Viestinvälitysarkkitehtuurissa periaate on sama, mutta toteutus on koko ohjelman laajuinen. Tällä hetkellä ohjelmassa toteutetussa viestinvälityksessä kulkee tietoa vain sulkemisesta. Toiminnan laajentamisen mahdollisuutta kannattaa tutkia sekä sitä, mitä rakenteita sillä voitaisiin korvata.

Kehitystoiminnan runkona on siinä käytettävä prosessimalli. Prosessimalleja on erilaisia, mutta pienelle tiimille sopii evoluutiopohjainen – malli. Sen iteraatioissa laajenevan ohjelman kehittäminen mahdollistaa ominaisuuksien lisäämisen ilman laajaa määrittelyä, joka muuten vaatii suurta työpanosta ilman tuloksia. Evoluutiomallin lisäksi voi hyödyntää osia komponentti -mallista, jossa ohjelmasta pystytään vaihtamaan kerralla suurempia osia.

Lähdekoodin laatu heikkenee siihen kohdistuvien muutosten seurauksesta. Syynä eivät ole pelkästään yleisestä linjasta poikkeavat toteutukset, vaan myös puutteellinen dokumentaatio. Se vaikuttaa koodin ymmärrettävyyteen, koska silloin tieto kyseisenlaisiin ratkaisuihin päätymisistä saattaa kadota. Tietomäärittelyiden avulla ohjelman tallentaman tiedon merkitys ja käsittely dokumentoidaan. Itse tuotettu lähdekoodi ei ole ainoa osa, joka vaatii ylläpitoa, vaan myös ulkoiset osat vaativat sitä. Niiden ylläpitämiseksi on tehtävä tarkastuksia säännöllisesti päivitysten osalta.

Uusia ominaisuuksia lisättäessä on oltava ymmärrystä siitä, mihin kyseinen muutos vaikuttaa ja kuinka paljon se vie aikaa. Kestosta on tiedettävä, viivästyttääkö ominaisuuden lisääminen version julkaisua. Tärkeimpänä dokumentaationa ominaisuuksia lisättäessä on muutospyyntödokumentti, johon merkitään ominaisuuden ja aikataulutuksen lisäksi tieto muuttuneista tietokentistä. Tietomuutoksien hallittavuutta kannattaa pyrkiä lisäämään kehityksessä. Päivityksissä muuttunutta tietoa pyritään hallitsemaan, mutta eri ohjelmatasojen välillä täytyy mahdollistaa tiedon siirrettävyys. Haasteeseen on perehdyttävä tämän työn ulkopuolella tulevaisuudessa.

Uudistaminen on poikkeuksellista toimintaa, johon ryhdyttäessä on oltava selkeät perusteet ja saatava hyöty selvillä. Riskikartoitusta on tehtävä ja täydennettävä läpi toimenpiteen, jotta ongelmiin voidaan reagoida nopeasti. Aiempaa dokumentaatiota tai lähdekoodia tutkimalla on hankittava ymmärrys toiminnasta. Ilman tekijän ymmärrystä alkuperäisestä kohteesta on mahdollista, että ohjelmaan kertynyttä ymmärrystä sovellusalasta katoaa. Uudistaminen itsessään on jo riski, mutta siihen sidottavat henkilöresurssit tekevät siitä pienelle tiimille vielä haastavamman. Uudistamista on toteutettava siten, että siihen sidottuja resursseja voidaan siirtää muihin projekteihin tarpeen mukaan väliaikaisesti.

Työkalujen tarkoituksena on tehtävien automatisoiminen, jotta inhimilliset virheet vähenisivät. Ohjeistusta niiden käytöstä on lisättävä, jotta automaatioastetta saadaan nostettua. Työkalujen ei tarvitse olla ulkoisia, vaan niitä voidaan tehdä myös sisäiseen käyttöön tarpeen vaatiessa.

Lähdekoodia analysoimalla saadaan selville rakenteiden lisäksi tietoa sen suorituksen aikaisista ominaisuuksista. Profiloijan hyödyntämiseksi on tehtävä käyttösuunnitelma, johon sisältyy myös toimintaohjeet.

Tietokannan käyttö saattaa muuttua asiakaspäässä pullonkaulaksi, jota kehitysvaiheessa on vaikea huomata. Kyselyjä analysoimalla on mahdollista löytää huonosti toimivat kyselyt tai puuttuvat indeksit.

Ohjelmasta saatavia raportteja voidaan monipuolistaa interaktiivisilla toiminnoilla. Sillä saadaan raporteille lisää toiminnallisuutta, jolla käyttäjä pystyy havainnoimaan paremmin sitä, mistä esitettävä tieto muodostuu. Raportoinnissa on muistettava, että sen pääasiallisena tarkoituksena ei ole toimia käyttöliittymänä.

Konfiguraation hallintaa käytetään järjestelmällisenä tapana luoda mahdollisimman ehjä tuote monenlaisista eri osista, joita sen muodostamiseksi vaaditaan. Se on lisäksi joukko käytäntöjä, joilla halutunlaisesta lopputuloksesta pystytään varmistumaan. Konfiguraation hallinta on hyödyllinen pienillekin tiimeille, jotta työn panokset pystytään keskittämään vain tarpeellisiin asioihin.

Hajautetulla versiohallinnalla voidaan toteuttaa versiointia aivan kuten keskitetylläkin. Hajautettujen etuna on se, että niissä ei ole tarvetta omalle palvelimelleen. Käyttämällä hajautettua versiohallintaa on mahdollista luoda uudenlaisia toimintamalleja versiointiin. Uusia toimintamalleja on edelleen arvioitava vastaan tulevien tilanteiden perusteella ja mukautettava niistä saadulla tiedolla.

Kaikki ongelmat, ominaisuudet ja muutokset on kirjattava järjestelmään seurannan vuoksi. Linkittämällä tehdyt muutokset tehtävienkirjaukseen pystytään löytämään nopeasti varsinaiset muutokset myös jälkikäteen.

Tässä työssä saadut tulokset tehostavat erityisesti pienen tiimin ohjelmiston kehittämistoimintaa pitkällä aikavälillä. Tuloksia voidaan hyödyntää myös suuremmissa yksiköissä. Silloin on kuitenkin otettava huomioon prosessimalli ja dokumentaation laatuvaatimukset. Kehittämällä tietomäärittelyjen dokumentaatiota pystytään tekniikasta riippumatta siirtämään tietämystä toisenlaiseen työskentely-ympäristöön. Tässä työssä monipuolisesti esitettyä modulaarisuuden parantamista voidaan tulevaisuudessa hyödyntää pitkän elinkaaren tuotteen kehittämisessä.

LÄHTEET

[1] M. Harsu, Ohjelmien ylläpito ja uudistaminen, Helsinki: Talentum, 2003, p. 292.

[2] I. Sommerville, Software Engineering, 8th toim., New York: Addison-Wesley, 2006, p. 840.

[3] K. Koskimies ja T. Mikkonen, Ohjelmistoarkkitehtuurit, Helsinki: Talentum, 2005, p. 250.

[4] F. Martin, ”AnemicDomainModel,” [Online]. Available: http://www.martinfowler.com/bliki/AnemicDomainModel.html. [Haettu 7 12 2011].

[5] Elmasari ja Navathe, Fundamentals of Database Systems, 5. painos toim., Reading (MA): Addison-Wesley, 2007, p. 1123.

[6] W. M. Osborne, Building Maintainable Software: Handbook of Systems Management (Development and Support), Auerbach Publishers, 1989.

[7] H. Dorota ja A. Kolowa, Automated Defect Prevention Best, New Jersey: John Wiley & Sons, Inc., 2007.

[8] J. Spolsky, ”Joel On Software,” 06 04 2000. [Online]. Available: http://www.joelonsoftware.com/articles/fog0000000069.html. [Haettu 30 10 2012].

[9] Microsoft, ”Using Statistics to Improve Query Performance,” 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms190397.aspx. [Haettu 7 Helmikuu 2012].

[10] Microsoft, ”MSDN,” 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms190439%28v=sql.105%29.aspx. [Haettu 7 Helmikuu 2012].

[11] Fast Reports Inc., ”FastReport 4.6 Programmer's manual,” Fast Reports Inc., 2008.

[12] I. Haikala ja J. Märijärvi, Ohjelmistotuotanto, 8. painos toim., Pieksämäki: Talentum Media Oy, 2002.

[13] A. Abran ja J. W. Moore, ”Guide to the Software Engineering Body of Knowledge,” IEEE Computer Society, Los Alamitos, 2004.

[14] IEEE Std 982.1-2005, ”IEEE Standard Dictionary of Measures of the Software Aspects of Dependability,” IEEE Computer Society, New York, 2005.

[15] P. C. e. al, ”Version Management with CVS,” 2008.

[16] Kiln, ”Kiln Support,” [Online]. Available: http://kiln.stackexchange.com/questions/1873/how-do-you-use-bfiles/1874#1874. [Haettu 13.5.2012 Toukokuu 2012].

[17] Flyspray, ”Flyspray - The Bug Killer!,” [Online]. Available: http://flyspray.org/start. [Haettu 21 Toukokuu 2012].

[18] CnPack, ”CnPack,” [Online]. Available: http://www.cnpack.org. [Haettu 21 1 2012].

[19] GExperts, ”GExperts,” [Online]. Available: http://www.gexperts.org/. [Haettu 21 1 2012].

[20] B. O'Sullivan, ”Mercurial: The Definitive Guide,” 2009. [Online]. Available: http://hgbook.red-bean.com. [Haettu 2 Toukokuu 2012].

[21] H. J. Ferdinand, ”Software Requirements: Update, Upgrade, Redesign Towards a Theory of Requirements Change,” Haarlem, 2006.

[22] Microsoft, ”What is Transaction?,” 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx. [Haettu 13 Helmikuu 2013].